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

Draft Technical Standard: System Interfaces, Issue 7

Prepared by the Austin Group (www.opengroup.org/austin)

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. This standard is intended to be used by both applications developers and system implementors and comprises four major components (each in an associated volume):

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

The following areas are outside the scope of this standard:

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

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

Keywords

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

Feedback

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

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

List of Tables

Foreword

Structure of the Standard

Notes to Reviewers

This section with side shading will not appear in the final copy. - Ed.

This section will be completed in a later draft.

Introduction

Note: This introduction is not part of IEEE Std 1003.1-200x, Standard for Information Technology — Portable Operating System Interface (POSIX).

This draft 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.[1]

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 is to develop and maintain the core open systems interfaces that are the POSIX® 1003.1 (and former 1003.2) standards, ISO/IEC 9945 Parts 1 to 4, and the core of the Single UNIX Specification.

The approach to specification development has been one of "write once, adopt everywhere", with the deliverables being a set of specifications that carry the IEEE POSIX designation, The Open Group's Technical Standard designation, and an ISO/IEC designation.

This 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 www.opengroup.org/austin.

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,[2] 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

---

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

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

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.

- 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—IEEE Std 1003.1-1988—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 2001 Edition of this standard and its technical corrigenda have have consolidated this consensus, and this revision reflects the significantly increased level of consensus arrived at since the original versions. 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 standard is specifically not a codification of a particular vendor's product.

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

- Minimal Changes to Existing Application Code

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

This Standard

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

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

This Volume

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

This volume is structured as follows:

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

Comprehensive references are available in the index.

Typographical Conventions The following typographical conventions are used throughout this standard. In the text, this standard is referred to as IEEE Std 1003.1-200x, which is technically identical to The Open Group Base Specifications, Issue 7.

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

Reference	Example	Notes
C-Language Data Structure	**aiocb**	
C-Language Data Structure Member	*aio_lio_opcode*	
C-Language Data Type	**long**	
C-Language External Variable	*errno*	
C-Language Function	*system*()	
C-Language Function Argument	*arg1*	
C-Language Function Family	*exec*	
C-Language Header	**<sys/stat.h>**	
C-Language Keyword	**return**	
C-Language Macro with Argument	*assert*()	
C-Language Macro with No Argument	INET_ADDRSTRLEN	
C-Language Preprocessing Directive	**#define**	
Commands within a Utility	**a**, **c**	
Conversion Specification, Specifier/Modifier Character	%A, g, E	1
Environment Variable	*PATH*	
Error Number	[EINTR]	
Example Output	**Hello, World**	
Filename	**/tmp**	
Literal Character	'c','\r','\'	2
Literal String	"abcde"	2

Reference	Example	Notes
Optional Items in Utility Syntax	**[ ]**	
Parameter	*<directory pathname>*	
Special Character	<newline>	3
Symbolic Constant	_POSIX_VDISABLE	
Symbolic Limit, Configuration Value	{LINE_MAX}	4
Syntax	`#include <sys/stat.h>`	
User Input and Example Code	`echo Hello, World`	5
Utility Name	*awk*	
Utility Operand	*file_name*	
Utility Option	**–c**	
Utility Option with Option-Argument	**–w** *width*	

Notes:

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

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

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

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

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

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

Notes:

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

Participants

IEEE Std 1003.1-200x was prepared by the Austin Group, sponsored by the Portable Applications Standards Committee of the IEEE Computer Society, The Open Group, and ISO/SC22.

Notes to Reviewers

This section with side shading will not appear in the final copy. - Ed.

This section will be completed once the standard is approved.

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.

Boundaryless Information Flow™ and TOGAF™ are trademarks and Motif®, Making Standards Work®, OSF/1®, The Open Group®, UNIX®, and the "X" device are registered trademarks of The Open Group in the United States and other countries.

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

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

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

Acknowledgements

The contributions of the following organizations to the development of IEEE Std 1003.1-200x 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.
- ISO/IEC JTC 1/SC 22/WG 14 C Language Committee

This standard was prepared by the Austin Group, a joint working group of the IEEE, The Open Group, and ISO/IEC JTC 1/SC 22.

Referenced Documents

Normative References

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

Informative References

The following documents are referenced in this standard:

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

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

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

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

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

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

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

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

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

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

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

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

Referenced Documents

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

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

IETF RFC 791
Internet Protocol, Version 4 (IPv4), September 1981.

IETF RFC 819
The Domain Naming Convention for Internet User Applications, Z. Su, J. Postel, August 1982.

IETF RFC 822
Standard for the Format of ARPA Internet Text Messages, D.H. Crocker, August 1982.

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

IETF RFC 920
Domain Requirements, J. Postel, J. Reynolds, October 1984.

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

IETF RFC 922
Broadcasting Internet Datagrams in the Presence of Subnets, J. Mogul, October 1984.

IETF RFC 1034
Domain Names — Concepts and Facilities, P. Mockapetris, November 1987.

IETF RFC 1035
Domain Names — Implementation and Specification, P. Mockapetris, November 1987.

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

IETF RFC 1886
DNS Extensions to Support Internet Protocol, Version 6 (IPv6), C. Huitema, S. Thomson, December 1995.

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

IETF RFC 2181
Clarifications to the DNS Specification, R. Elz, R. Bush, July 1997.

IETF RFC 2373
Internet Protocol, Version 6 (IPv6) Addressing Architecture, S. Deering, R. Hinden, July 1998.

IETF RFC 2460
Internet Protocol, Version 6 (IPv6), S. Deering, R. Hinden, December 1998.

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

ISO C (1990)

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

ISO 2375: 1985

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

ISO 8652: 1987

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

ISO/IEC 1539: 1990

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

ISO/IEC 4873: 1991

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

ISO/IEC 6429: 1992

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

ISO/IEC 6937: 1994

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

ISO/IEC 8802-3: 1996

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

ISO/IEC 8859

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

Part 1: Latin Alphabet No. 1
Part 2: Latin Alphabet No. 2
Part 3: Latin Alphabet No. 3
Part 4: Latin Alphabet No. 4
Part 5: Latin/Cyrillic Alphabet
Part 6: Latin/Arabic Alphabet
Part 7: Latin/Greek Alphabet
Part 8: Latin/Hebrew Alphabet
Part 9: Latin Alphabet No. 5
Part 10: Latin Alphabet No. 6
Part 11: Latin/Thai Alphabet
Part 13: Latin Alphabet No. 7
Part 14: Latin Alphabet No. 8
Part 15: Latin Alphabet No. 9
Part 16: Latin Alphabet No. 10

ISO POSIX-1: 1996

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

Referenced Documents

ISO POSIX-2: 1993

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

Issue 1

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

Issue 2

X/Open Portability Guide, January 1987:

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

Issue 3

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

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

Issue 4

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

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

Issue 4, Version 2

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

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

Issue 5

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

- System Interface Definitions (XBD), Issue 5 (ISBN: 1-85912-186-1, C605)
- Commands and Utilities (XCU), Issue 5 (ISBN: 1-85912-191-8, C604)

- System Interfaces and Headers (XSH), Issue 5 (ISBN: 1-85912-181-0, C606)

Issue 6

Technical Standard, April 2004, published by The Open Group:

- Base Definitions (XBD), Issue 6 (ISBN: 1-931624-43-7, C046)
- System Interfaces (XSH), Issue 6 (ISBN: 1-931624-44-5, C047)
- Shell and Utilities (XCU), Issue 6 (ISBN: 1-931624-45-3, C048)

Knuth Article

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

KornShell

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

MSE Working Draft

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

POSIX.0:1995

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

POSIX.1:1988

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

POSIX.1:1990

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

POSIX.1a

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

POSIX.1d:1999

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

POSIX.1g:2000

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

POSIX.1j:2000

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

Referenced Documents

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

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

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

POSIX.13:-1998
IEEE Std 1003.13: 1998, IEEE Standard for Information Technology — Standardized Application Environment Profile (AEP) — POSIX Realtime Application Support.

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

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

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

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

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

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

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

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

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

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

X/Open Curses, Issue 4, Version 2

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

Yacc

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

Source Documents

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

AIX 3.2 Manual

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

OSF/1

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

OSF AES

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

System V Release 2.0

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

— UNIX System V Release 2.0 Programming Guide (April 1984 - Issue 2).

System V Release 4.2

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

Chapter 1

Introduction

1.1 Scope

The scope of IEEE Std 1003.1-200x is described in the Base Definitions volume of IEEE Std 1003.1-200x.

1.2 Conformance

Conformance requirements for IEEE Std 1003.1-200x are defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 2, Conformance.

1.3 Normative References

Normative references for IEEE Std 1003.1-200x are defined in the Base Definitions volume of IEEE Std 1003.1-200x.

1.4 Change History

Change history is described in the Rationale (Informative) volume of IEEE Std 1003.1-200x, and in the CHANGE HISTORY section of reference pages.

1.5 Terminology

This section appears in the Base Definitions volume of IEEE Std 1003.1-200x, but is repeated here for convenience:

For the purposes of IEEE Std 1003.1-200x, 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 1003.1-200x. 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 1003.1-200x but is selected by an implementor. The value or behavior may vary among implementations that conform to IEEE Std 1003.1-200x. 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 1003.1-200x. 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 1003.1-200x, 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 1003.1-200x, 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 1003.1-200x 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 1003.1-200x. 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 1003.1-200x 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 1003.1-200x. An application should not rely on the existence or validity of the value or behavior. An application that relies on any particular value or behavior cannot be assured to be portable across conforming implementations.

1.6 Definitions

Concepts and definitions are defined in the Base Definitions volume of IEEE Std 1003.1-200x.

1.7 Relationship to Other Formal Standards

Great care has been taken to ensure that this volume of IEEE Std 1003.1-200x 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 1003.1-200x. Some functions and headers included within this volume of IEEE Std 1003.1-200x 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 (on page 3)). Any conflict between this volume of IEEE Std 1003.1-200x and the ISO C standard is unintentional.

This volume of IEEE Std 1003.1-200x 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 1003.1-200x and functions in the System Interfaces volume of IEEE Std 1003.1-200x describe functionality that might not be fully portable to systems meeting the requirements for POSIX conformance (see the Base Definitions volume of IEEE Std 1003.1-200x, 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 (on page 3)). 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 1003.1-200x, 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.

BE 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.

CD 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.

CPT Process CPU-Time Clocks

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 1003.1-200x, Section 1.5.2, Margin Code Notation.

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.

MLR Range 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 MLR margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MLR margin legend.

MON Monotonic Clock

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

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

MSG Message Passing

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

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

MX IEC 60559 Floating-Point

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

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

OB Obsolescent

The functionality described may be withdrawn in a future version of this volume of IEEE Std 1003.1-200x. Strictly Conforming POSIX Applications and Strictly Conforming XSI Applications shall not use obsolescent features.

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

OF Output Format Incompletely Specified

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

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

OH **Optional Header**

In the SYNOPSIS section of some interfaces in the System Interfaces volume of IEEE Std 1003.1-200x an included header is marked as in the following example:

OH
```
#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.

PIO **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.

PS **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.

RPI **Robust Mutex 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 RPI margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the RPI margin legend.

RPP **Robust Mutex 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 RPP margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the RPP margin legend.

RS **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.

SD **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.

SHM **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 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.

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. This functionality is dependent on support for the Trace option. 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.

TPI **Non-Robust Mutex 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 **Non-Robust Mutex 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. This functionality is dependent on support for the Trace option. 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. This functionality is dependent on support for the Trace option. 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.

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 Size 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 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.

UU **UUCP Utilities**
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

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

XSI **X/Open System Interfaces**
The functionality described is part of the X/Open Systems Interfaces option. Functionality marked XSI is an extension to the ISO C standard. Application writers may confidently make use of such extensions on all systems supporting the X/Open System Interfaces option.

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 1003.1-200x, Section 3.439, XSI.

XSR **XSI STREAMS**
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

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

1.9 Format of Entries

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

NAME

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

SYNOPSIS

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

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

DESCRIPTION

This section describes the functionality of the function or header.

RETURN VALUE

This section indicates the possible return values, if any.

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

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

ERRORS

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

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

EXAMPLES

This section is informative.

This section gives examples of usage, where appropriate. In the event of conflict between an example and a normative part of this volume of IEEE Std 1003.1-200x, 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 1003.1-200x, 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 1003.1-200x 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

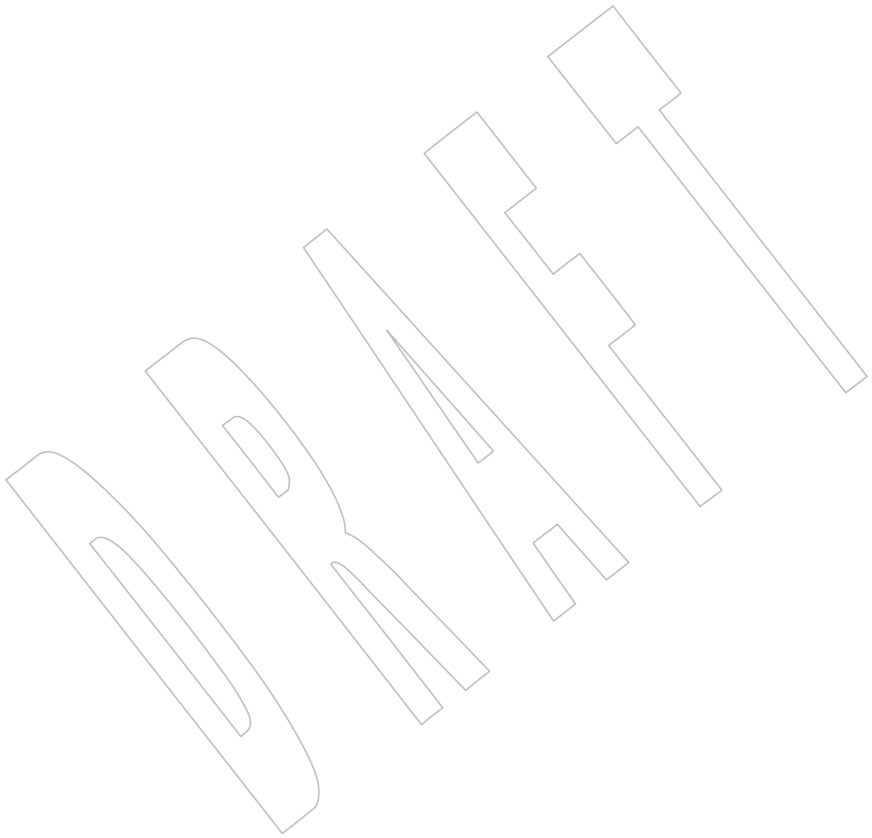
This section is informative.

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.



418 *Chapter 2*

419 # *General Information*

420 This chapter covers information that is relevant to all the functions specified in Chapter 3 and
421 the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers.

422 ## 2.1 Use and Implementation of Functions

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

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

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

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

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

444 5. If a function that accepts a variable number of arguments is not declared (explicitly or by
445 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 1003.1-200x are defined in headers (see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers). Some of those headers could also define symbols other than those defined by IEEE Std 1003.1-200x, potentially conflicting with symbols used by the application. Also, IEEE Std 1003.1-200x 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 1003.1-200x, and other standards may define additional feature test macros.

In the compilation of an application that **#define**s a feature test macro specified by IEEE Std 1003.1-200x, no header defined by IEEE Std 1003.1-200x 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 1003.1-200x, and when this feature test macro is defined to have the value 200xxxL:

1. All symbols required by IEEE Std 1003.1-200x to appear when the header is included shall be made visible.
2. Symbols that are explicitly permitted, but not required, by IEEE Std 1003.1-200x 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 1003.1-200x to be in that header shall not be made visible, except when enabled by another feature test macro.

Identifiers in IEEE Std 1003.1-200x may only be undefined using the **#undef** directive as described in Section 2.1 or Section 2.2.2 (on page 15). These **#undef** directives shall follow all **#include** directives of any header in IEEE Std 1003.1-200x.

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*

XSI An XSI-conforming application should ensure that the feature test macro _XOPEN_SOURCE is defined with the value 700 before inclusion of any header. This is needed to enable the functionality described in Section 2.2.1.1 and to ensure that the XSI option is enabled.

Since this volume of IEEE Std 1003.1-200x is aligned with the ISO C standard, and since all functionality enabled by _POSIX_C_SOURCE set equal to 200xxxL is enabled by _XOPEN_SOURCE set equal to 700, there should be no need to define _POSIX_C_SOURCE if _XOPEN_SOURCE is so defined. Therefore, if _XOPEN_SOURCE is set equal to 700 and _POSIX_C_SOURCE is set equal to 200xxxL, the behavior is the same as if only _XOPEN_SOURCE is defined and set equal to 700. However, should _POSIX_C_SOURCE be set

to a value greater than 200xxxL, the behavior is unspecified.

If _XOPEN_SOURCE is defined with the value 700 and _POSIX_C_SOURCE is undefined before inclusion of any header, then the header may define the _POSIX_C_SOURCE macro with the value 200xxxL.

2.2.2 The Name Space

All identifiers in this volume of IEEE Std 1003.1-200x, except *environ*, are defined in at least one of the headers, as shown in the Base Definitions volume of IEEE Std 1003.1-200x, 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.

XSI

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 1003.1-200x and other POSIX standards. Implementations may add symbols to the headers shown in the following table, provided the identifiers for those symbols either:

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

Symbols that use the reserved prefix _POSIX_ may be made visible by implementations in any header defined by IEEE Std 1003.1-200x.

	Header	Prefix	Suffix	Complete Name
	<aio.h>	aio_, lio_, AIO_, LIO_		
	<arpa/inet.h>	in_, inet_		
	<ctype.h>	to[a-z], is[a-z]		
	<dirent.h>	d_		
	<errno.h>	E[0-9], E[A-Z]		
	<fcntl.h>	l_		
	<fenv.h>	FE_[A-Z]		
	<glob.h>	gl_		
	<grp.h>	gr_		
	<inttypes.h>			int[0-9a-z_]*_t, uint[0-9a-z_]*_t
	<limits.h>		_MAX, _MIN	
	<locale.h>	LC_[A-Z]		
MSG	**<mqueue.h>**	mq_, MQ_		
XSI	**<ndbm.h>**	dbm_		
	<netdb.h>	ai_, h_, n_, p_, s_		
	<net/if.h>	if_		
	<netinet/in.h>	in_, ip_, s_, sin_		
IP6		in6_, s6_, sin6_		
XSI	**<poll.h>**	pd_, ph_, ps_		
	<pthread.h>	pthread_, PTHREAD_		
	<pwd.h>	pw_		
	<regex.h>	re_, rm_		
PS	**<sched.h>**	sched_, SCHED_		
	<semaphore.h>	sem_, SEM_		
	<signal.h>	sa_, si_, SI_, SIG[A-Z], SIG_[A-Z], sigev_, SIGEV_, sival_, uc_,		
XSI		ss_, sv_		
OB XSR	**<stropts.h>**	bi_, ic_, l_, sl_, str_		
	<stdint.h>			int[0-9a-z_]*_t, uint[0-9a-z_]*_t
	<stdlib.h>	str[a-z]		
	<string.h>	str[a-z], mem[a-z], wcs[a-z]		
XSI	**<sys/ipc.h>**	ipc_		key, pad, seq
	<sys/mman.h>	shm_, MAP_, MCL_, MS_, PROT_		
XSI	**<sys/msg.h>**	msg		msg
XSI	**<sys/resource.h>**	rlim_, ru_		
	<sys/select.h>	fd_, fds_, FD_		
XSI	**<sys/sem.h>**	sem		sem
XSI	**<sys/shm.h>**	shm		
	<sys/socket.h>	ss_, sa_, if_, ifc_, ifru_, infu_, ifra_, msg_, cmsg_, l_		
	<sys/stat.h>	st_		
	<sys/statvfs.h>	f_		
XSI	**<sys/time.h>**	fds_, it_, tv_, FD_		
	<sys/times.h>	tms_		

	Header	Prefix	Suffix	Complete Name
XSI	**<sys/uio.h>**	iov_		UIO_MAXIOV
	<sys/un.h>	sun_		
	<sys/utsname.h>	uts_		
XSI	**<sys/wait.h>**	si_, W[A-Z], P_		
	<termios.h>	c_		
	<time.h>	tm_ clock_, timer_, it_, tv_, CLOCK_, TIMER_		
XSI	**<ulimit.h>**	UL_		
	<utime.h>	utim_		
XSI	**<utmpx.h>**	ut_	_LVL, _TIME, _PROCESS	
	<wchar.h>	wcs[a-z]		
	<wctype.h>	is[a-z], to[a-z]		
	<wordexp.h>	we_		
	ANY header		_t	

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

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

	Header	Prefix
	<dlfcn.h>	RTLD_
	<fcntl.h>	F_, O_, S_, SEEK_
XSI	**<fmtmsg.h>**	MM_
	<fnmatch.h>	FNM_
XSI	**<ftw.h>**	FTW
	<glob.h>	GLOB_
	<inttypes.h>	PRI[Xa-z], SCN[Xa-z]
	<math.h>	FP_[A-Z]
XSI	**<ndbm.h>**	DBM_
	<net/if.h>	IF_
	<netinet/in.h>	IMPLINK_, IN_, INADDR_, IP_, IPPORT_, IPPROTO_, SOCK_
IP6		IPV6_, IN6_
	<netinet/tcp.h>	TCP_
	<nl_types.h>	NL_
	<poll.h>	POLL
	<regex.h>	REG_
	<signal.h>	BUS_, CLD_, FPE_, ILL_, SA_, SEGV_, SI_, SIG_[0-9a-z_],
XSI		SS_, SV_, TRAP_
OB XSR		POLL_
CX	**<stdio.h>**	SEEK_
OB XSR	**<stropts.h>**	FLUSH[A-Z], I_, M_, MUXID_R[A-Z], S_, SND[A-Z], STR
XSI	**<syslog.h>**	LOG_
XSI	**<sys/ipc.h>**	IPC_
XSI	**<sys/mman.h>**	PROT_, MAP_, MS_
XSI	**<sys/msg.h>**	MSG[A-Z], MSG_[A-Z]
XSI	**<sys/resource.h>**	PRIO_, RLIM_, RLIMIT_, RUSAGE_
XSI	**<sys/sem.h>**	SEM_
XSI	**<sys/shm.h>**	SHM[A-Z], SHM_[A-Z]
XSI	**<sys/socket.h>**	AF_, CMSG_, MSG_, PF_, SCM_, SHUT_, SO
	<sys/stat.h>	S_
	<sys/statvfs.h>	ST_
XSI	**<sys/time.h>**	FD_, ITIMER_
XSI	**<sys/uio.h>**	IOV_
XSI	**<sys/wait.h>**	BUS_, CLD_, FPE_, ILL_, POLL_, SEGV_, SI_, TRAP_
	<termios.h>	V, I, O, TC, B[0-9] (See below.)
	<unistd.h>	SEEK_
	<wordexp.h>	WRDE_

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

INT[0-9A-Za-z_]*_MIN
INT[0-9A-Za-z_]*_MAX
INT[0-9A-Za-z_]*_C
UINT[0-9A-Za-z_]*_MIN
UINT[0-9A-Za-z_]*_MAX
UINT[0-9A-Za-z_]*_C

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.

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

645 CBAUD	EXTB	VDSUSP
646 DEFECHO	FLUSHO	VLNEXT
647 ECHOCTL	LOBLK	VREPRINT
648 ECHOKE	PENDIN	VSTATUS
649 ECHOPRT	SWTCH	VWERASE
650 EXTA	VDISCARD	

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

652 1. With the exception of identifiers beginning with the prefix _POSIX_, all identifiers that
653 begin with an underscore and either an uppercase letter or another underscore are always
654 reserved for any use by the implementation.

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

657 3. All identifiers in the table below are reserved for use as identifiers with external linkage.
658 Some of these identifiers do not appear in this volume of IEEE Std 1003.1-200x, but are
659 reserved for future use by the ISO C standard.

660 _Exit	atof	catanhl	cimag	cosl
661 abort	atoi	catanl	cimagf	cpow
662 abs	atol	cbrt	cimagl	cpowf
663 acos	atoll	cbrtf	clearerr	cpowl
664 acosf	bsearch	cbrtl	clgamma	cproj
665 acosh	cabs	ccos	clgammaf	cprojf
666 acoshf	cabsf	ccosf	clgammal	cprojl
667 acoshl	cabsl	ccosh	clock	creal
668 acosl	cacos	ccoshf	clog	crealf
669 acosl	cacosf	ccoshl	clog10	creall
670 asctime	cacosh	ccosl	clog10f	csin
671 asin	cacoshf	ceil	clog10l	csinf
672 asinf	cacoshl	ceilf	clog1p	csinh
673 asinh	cacosl	ceilf	clog1pf	csinhf
674 asinhf	calloc	ceill	clog1pl	csinhl
675 asinhl	carg	ceill	clog2	csinl
676 asinl	cargf	cerf	clog2f	csqrt
677 asinl	cargl	cerfc	clog2l	csqrtf
678 atan	casin	cerfcf	clogf	csqrtl
679 atan2	casinf	cerfcl	clogl	ctan
680 atan2f	casinh	cerff	conj	ctanf
681 atan2l	casinhf	cerfl	conjf	ctanl
682 atanf	casinhl	cexmp1	conjl	ctgamma
683 atanf	casinl	cexmp1f	copysign	ctgammaf
684 atanh	catan	cexmp1l	copysignf	ctgammal
685 atanh	catanf	cexp	copysignl	ltime
686 atanhf	catanh	cexp2	cos	difftime
687 atanhl	catanh	cexp2f	cosf	div
688 atanl	catanhf	cexp2l	cosh	erfcf
689 atanl	catanhf	cexpf	coshf	erfcl
690 atexit	catanhl	cexpl	coshl	erff

erfl	fopen	localtime	putchar	swprintf
errno	fprintf	log	puts	swscanf
exit	fputc	log10	putwc	system
exp	fputs	log10f	putwchar	tan
exp2	fputwc	log10l	qsort	tanf
exp2f	fputws	log1p	raise	tanh
exp2l	fread	log1pf	rand	tanhf
expf	free	log1pl	realloc	tanhl
expl	freopen	log2	remainderf	tanl
expm1	frexp	log2f	remainderl	tgamma
expm1f	frexpf	log2l	remove	tgammaf
expm1l	frexpl	logb	remquo	tgammal
fabs	fscanf	logbf	remquof	time
fabsf	fseek	logbl	remquol	tmpfile
fabsl	fsetpos	logf	rename	tmpnam
fclose	ftell	logl	rewind	to[a-z]*
fdim	fwide	longjmp	rint	trunc
fdimf	fwprintf	lrint	rintf	truncf
fdiml	fwrite	lrintf	rintl	truncl
feclearexcept	fwscanf	lrintl	round	ungetc
fegetenv	getc	lround	roundf	ungetwc
fegetexceptflag	getchar	lroundf	roundl	va_end
fegetround	getenv	lroundl	scalbln	vfprintf
feholdexcept	gets	malloc	scalblnf	vfscanf
feof	getwc	mblen	scalblnl	vfwprintf
feraiseexcept	getwchar	mbrlen	scalbn	vfwscanf
ferror	gmtime	mbrtowc	scalbnf	vprintf
fesetenv	hypotf	mbsinit	scalbnl	vscanf
fesetexceptflag	hypotl	mbsrtowcs	scanf	vsprintf
fesetround	ilogb	mbstowcs	setbuf	vsscanf
fetestexcept	ilogbf	mbtowc	setjmp	vswprintf
feupdateenv	ilogbl	mem[a-z]*	setlocale	vswscanf
fflush	imaxabs	mktime	setvbuf	vwprintf
fgetc	imaxdiv	modf	signal	vwscanf
fgetpos	is[a-z]*	modff	sin	wcrtomb
fgets	isblank	modfl	sinf	wcs[a-z]*
fgetwc	iswblank	nan	sinh	wcstof
fgetws	labs	nanf	sinhf	wcstoimax
floor	ldexp	nanl	sinhl	wcstold
floorf	ldexpf	nearbyint	sinl	wcstoll
floorl	ldexpl	nearbyintf	sprintf	wcstoull
fma	ldiv	nearbyintl	sqrt	wcstoumax
fmaf	ldiv	nextafterf	sqrtf	wctob
fmal	lgammaf	nextafterl	sqrtl	wctomb
fmax	lgammal	nexttoward	srand	wctrans
fmaxf	llabs	nexttowardf	sscanf	wctype
fmaxl	llrint	nexttowardl	str[a-z]*	wcwidth
fmin	llrintf	perror	strtof	wmem[a-z]*
fminf	llrintl	pow	strtoimax	wprintf
fminl	llround	powf	strtold	wscanf
fmod	llroundf	powl	strtoll	
fmodf	llroundl	printf	strtoull	
fmodl	localeconv	putc	strtoumax	

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

746 4. All functions and external identifiers defined in the Base Definitions volume of
747 IEEE Std 1003.1-200x, Chapter 13, Headers are reserved for use as identifiers with external
748 linkage.

749 5. All the identifiers defined in this volume of IEEE Std 1003.1-200x that have external linkage
750 are always reserved for use as identifiers with external linkage.

751 No other identifiers are reserved.

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

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

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

765 2.3 Error Numbers

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

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

773 The value of *errno* should only be examined when it is indicated to be valid by a function's
774 return value. No function in this volume of IEEE Std 1003.1-200x shall set *errno* to zero. For each
775 thread of a process, the value of *errno* shall not be affected by function calls or assignments to
776 *errno* by other threads.

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

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

781 Implementations may support additional errors not included in this list, may generate errors
782 included in this list under circumstances other than those described here, or may contain
783 extensions or limitations that prevent some errors from occurring. The ERRORS section on each
784 reference page specifies whether an error shall be returned, or whether it may be returned.
785 Implementations shall not generate a different error number from the ones described here for
786 error conditions described in this volume of IEEE Std 1003.1-200x, but may generate additional
787 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 1003.1-200x is detected. The conformance document may also contain statements that one or more of the optional error conditions are not detected.

Certain threads-related functions are not allowed to return an error code of [EINTR]. Where this applies it is stated in the ERRORS section on the individual function pages.

The following symbolic names identify the possible error numbers, in the context of the functions specifically defined in this volume of IEEE Std 1003.1-200x; 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 1003.1-200x. The actual values are unspecified by this volume of IEEE Std 1003.1-200x.

[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]
OB XSR 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.

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

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

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

838 or:

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

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

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

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

850 [ECONNABORTED]
851 Connection aborted. The connection has been aborted.

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

856 [ECONNRESET]
857 Connection reset. The connection was forcibly closed by the peer.

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

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

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

866 [EDQUOT]
867 Reserved.

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

871 [EFAULT]
872 Bad address. The system detected an invalid address in attempting to use an argument of a
873 call. The reliable detection of this error cannot be guaranteed, and when not detected may
874 result in the generation of a signal, indicating an address violation, which is sent to the
875 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 IEEE Std 1003.1-200x, **<signal.h>**).

[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]
File descriptor value too large. An attempt was made to open a file descriptor with a value
XSI greater than or equal to {OPEN_MAX}, or greater than or equal to the soft limit RLIMIT_NOFILE for the process (if smaller than {OPEN_MAX}).

[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.

OB XSR [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.

[ENOMSG]
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 *setsockopt*( ) 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.

OB XSR [ENOSR]
No STREAM resources. Insufficient STREAMS memory resources are available to perform a STREAMS-related function. This is a temporary condition; it may be recovered from if other processes release resources.

OB XSR [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. A conforming implementation may assign the same values for [EOPNOTSUP] and [ENOTSUP].

[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.

OB XSR [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 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 30)). 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. 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 1003.1-200x under which the implementation generates signals.

2.4.2 Realtime Signal Generation and Delivery

This section describes functionality to support realtime signal generation and delivery.

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

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

The *sigev_notify* member specifies the notification mechanism to use when an asynchronous event occurs. This volume of IEEE Std 1003.1-200x defines the following values for the *sigev_notify* member:

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

SIGEV_SIGNAL — The signal specified in *sigev_signo* shall be generated for the process when the event of interest occurs. If the implementation supports the Realtime Signals Extension option and if the SA_SIGINFO flag is set for that signal number, then the signal shall be queued to the process and the value specified in *sigev_value* shall be the *si_value* component of the generated signal. If SA_SIGINFO is not set for that signal number, it is unspecified whether the signal is queued and what value, if any, is sent.

SIGEV_THREAD A notification function shall be called to perform notification.

An implementation may define additional notification mechanisms.

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

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

Member Type	Member Name	Description
int	*sival_int*	Integer signal value.
void*	*sival_ptr*	Pointer signal value.

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

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

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

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

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

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

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

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 1003.1-200x are specified under **<signal.h>**. 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. 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.

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

XSI 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,
XSI 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].

Any queued values pending shall be discarded and the resources used to queue them shall be released and made available to queue other signals.

Pointer to a Function

Catch signal.

On delivery of the signal, the receiving process is to execute the signal-catching function at the specified address. After returning from the signal-catching 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-catching function shall be entered as a C-language function call as follows:

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

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

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

where *func* is the specified signal-catching 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:

Member Type	Member Name	Description
int	*si_signo*	Signal number.
int	*si_code*	Cause of the signal.
union sigval	*si_value*	Signal value.

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.

MSG 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 *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-catching function for a SIGBUS, SIGFPE, SIGILL, or SIGSEGV signal that was not generated by *kill*(), *sigqueue*(), or *raise*().

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

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

When signal-catching functions are invoked asynchronously with process execution, the behavior of some of the functions defined by this volume of IEEE Std 1003.1-200x is unspecified

if they are called from a signal-catching function.

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

_Exit()	*faccessat*()	*link*()	*rmdir*()	*socket*()
_exit()	*fchmod*()	*linkat*()	*select*()	*socketpair*()
abort()	*fchmodat*()	*listen*()	*sem_post*()	*stat*()
accept()	*fchown*()	*lseek*()	*send*()	*symlink*()
access()	*fchownat*()	*lstat*()	*sendmsg*()	*symlinkat*()
aio_error()	*fcntl*()	*mkdir*()	*sendto*()	*sysconf*()
aio_return()	*fdatasync*()	*mkdirat*()	*setgid*()	*tcdrain*()
aio_suspend()	*fexecve*()	*mkfifo*()	*setpgid*()	*tcflow*()
alarm()	*fork*()	*mkfifoat*()	*setsid*()	*tcflush*()
bind()	*fpathconf*()	*mknodat*()	*setsockopt*()	*tcgetattr*()
cfgetispeed()	*fstat*()	*open*()	*setuid*()	*tcgetpgrp*()
cfgetospeed()	*fstatat*()	*openat*()	*shutdown*()	*tcsendbreak*()
cfsetispeed()	*fsync*()	*pathconf*()	*sigaction*()	*tcsetattr*()
cfsetospeed()	*ftruncate*()	*pause*()	*sigaddset*()	*tcsetpgrp*()
chdir()	*futimesat*()	*pipe*()	*sigdelset*()	*time*()
chmod()	*getegid*()	*poll*()	*sigemptyset*()	*timer_getoverrun*()
chown()	*geteuid*()	*posix_trace_event*()	*sigfillset*()	*timer_gettime*()
clock_gettime()	*getgid*()	*pselect*()	*sigismember*()	*timer_settime*()
close()	*getgroups*()	*raise*()	*signal*()	*times*()
connect()	*getpeername*()	*read*()	*sigpause*()	*umask*()
creat()	*getpgrp*()	*readlink*()	*sigpending*()	*uname*()
dup()	*getpid*()	*readlinkat*()	*sigprocmask*()	*unlink*()
dup2()	*getppid*()	*recv*()	*sigqueue*()	*unlinkat*()
execl()	*getsockname*()	*recvfrom*()	*sigset*()	*utime*()
execle()	*getsockopt*()	*recvmsg*()	*sigsuspend*()	*wait*()
execv()	*getuid*()	*rename*()	*sleep*()	*waitpid*()
execve()	*kill*()	*renameat*()	*sockatmark*()	*write*()

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 1003.1-200x shall behave as defined when called from or interrupted by a signal-catching function, with a single exception: when a signal interrupts an unsafe function and the signal-catching 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 1003.1-200x 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-catching function, the signal-catching function shall be invoked; in this case the original function is said to be "interrupted" by the signal. If the signal-catching 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

CX A stream is associated with an external file (which may be a physical device) or memory buffer
CX by "opening" a file or buffer. This 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.

CX A stream associated with a memory buffer shall have the same operations for text files that a stream associated with an external file would have. In addition, the stream orientation shall be determined in exactly the same fashion.

Input and output operations on a stream associated with a memory buffer by a call to *fmemopen*( ) shall be constrained by the implementation to take place within the bounds of the memory buffer. In the case of a stream opened by *open_memstream*( ) or *open_wmemstream*( ), the memory area shall grow dynamically to accommodate write operations as necessary. For output, data is moved from the buffer provided by *setvbuf*( ) to the memory stream during a flush or close operation.

2.5.1 Interaction of File Descriptors and Standard I/O Streams

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

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

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

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

The result of function calls involving any one handle (the "active handle") is defined elsewhere in this volume of IEEE Std 1003.1-200x, but if two or more handles are used, and any one of them is a stream, the application shall ensure that their actions are coordinated as described below. If this is not done, the result is undefined.

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

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

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

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

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

- If it is a file descriptor, no action is required.
- If the only further action to be performed on any handle to this open file descriptor is to close it, no action need be taken.
- If it is a stream which is unbuffered, no action need be taken.

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

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

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

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

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

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

Otherwise, the result is undefined.

For the second handle:

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

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

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

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

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

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

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 *freopen*( ) function or the *fwide*( ) function can otherwise alter the orientation of a stream.

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

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 *mbrtowc*() function, with the conversion state described by the
CX 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 *wcrtomb*() function, with the conversion state
CX 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 *mbrtowc*() function does not form a valid (generalized) character, or if the code value passed to the underlying *wcrtomb*() function does not correspond to a valid (generalized) character. The wide-character input/output functions and the byte input/output functions store the value of the macro [EILSEQ] in *errno* if and only if an encoding error occurs.

2.6 STREAMS

OB XSR STREAMS functionality is provided on implementations supporting the XSI STREAMS Option Group. The functionality described in this section 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 1003.1-200x. Creation of STREAMS modules is outside the scope of IEEE Std 1003.1-200x.

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

XSI This section describes extensions to support interprocess communication. The functionality described in this section shall be provided on implementations that support the XSI option (and the rest of this section is not further shaded).

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

IPC Functions		
msgctl( )	*semctl*( )	*shmctl*( )
msgget( )	*semget*( )	*shmdt*( )
msgrcv( )	*semop*( )	*shmget*( )
msgsnd( )	*shmat*( )	

Another interprocess communication facility is provided by functions in the Realtime Option Group; see Section 2.8 (on page 40).

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 1003.1-200x, **<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 1003.1-200x, **<sys/ipc.h>**) which are used in conjunction to determine whether or not read/write (read/alter for semaphores) permissions should be granted to processes using the IPC facilities. The *mode* member of the **ipc_perm** structure acts as a bit field which determines the permissions.

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

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

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

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

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 some 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 1003.1-200x, 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 1003.1-200x are portable, although some of the numeric parameters used by an implementation may have hardware dependencies.

2.8.1 Realtime Signals

See Section 2.4.2 (on page 29).

2.8.2 Asynchronous I/O

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 1003.1-200x, **<aio.h>** and has at least the following members:

Member Type	Member Name	Description
int	*aio_fildes*	File descriptor.
off_t	*aio_offset*	File offset.
volatile void*	*aio_buf*	Location of buffer.
size_t	*aio_nbytes*	Length of transfer.
int	*aio_reqprio*	Request priority offset.
struct sigevent	*aio_sigevent*	Signal number and value.
int	*aio_lio_opcode*	Operation to be performed.

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

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

If _POSIX_PRIORITIZED_IO and _POSIX_PRIORITY_SCHEDULING are defined, then asynchronous I/O is queued in priority order, with the priority of each asynchronous operation based on the current scheduling priority of the calling process. The *aio_reqprio* member can be used to lower (but not raise) the asynchronous I/O operation priority and is within the range zero through {AIO_PRIO_DELTA_MAX}, inclusive. Unless both _POSIX_PRIORITIZED_IO and _POSIX_PRIORITY_SCHEDULING are defined, the order of processing asynchronous I/O requests is unspecified. When both _POSIX_PRIORITIZED_IO and _POSIX_PRIORITY_SCHEDULING are defined, the order of processing of requests submitted by processes whose schedulers are not SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC is

unspecified. The priority of an asynchronous request is computed as (process scheduling priority) minus *aio_reqprio*. The priority assigned to each asynchronous I/O request is an indication of the desired order of execution of the request relative to other asynchronous I/O requests for this file. If _POSIX_PRIORITIZED_IO is defined, requests issued with the same priority to a character special file are processed by the underlying device in FIFO order; the order of processing of requests of the same priority issued to files that are not character special files is unspecified. Numerically higher priority values indicate requests of higher priority. The value of *aio_reqprio* has no effect on process scheduling priority. When prioritized asynchronous I/O requests to the same file are blocked waiting for a resource required for that I/O operation, the higher-priority I/O requests shall be granted the resource before lower-priority I/O requests are granted the resource. The relative priority of asynchronous I/O and synchronous I/O is implementation-defined. If _POSIX_PRIORITIZED_IO is defined, the implementation shall define for which files I/O prioritization is supported.

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

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

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

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

MLR 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.

ML | MLR 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 *fork*(), and all memory locks owned by a process are unlocked upon *exec* or process termination. Unmapping of an address range removes any memory locks established on that address range by this process.

1706 ### 2.8.3.2 *Memory Mapped Files*

1707 Range memory mapping operations are defined in terms of pages. Implementations may
1708 restrict the size and alignment of range mappings to be on page-size boundaries. The page size,
1709 in bytes, is the value of the configurable system variable {PAGESIZE}. If an implementation has
1710 no restrictions on size or alignment, it may specify a 1-byte page size.

1711 Memory mapped files provide a mechanism that allows a process to access files by directly
1712 incorporating file data into its address space. Once a file is mapped into a process address space,
1713 the data can be manipulated as memory. If more than one process maps a file, its contents are
1714 shared among them. If the mappings allow shared write access, then data written into the
1715 memory object through the address space of one process appears in the address spaces of all
1716 processes that similarly map the same portion of the memory object.

1717 SHM Shared memory objects are named regions of storage that may be independent of the file system
1718 and can be mapped into the address space of one or more processes to allow them to share the
1719 associated memory.

1720 SHM An *unlink*() of a file or *shm_unlink*() of a shared memory object, while causing the removal of
1721 the name, does not unmap any mappings established for the object. Once the name has been
1722 removed, the contents of the memory object are preserved as long as it is referenced. The
1723 memory object remains referenced as long as a process has the memory object open or has some
1724 area of the memory object mapped.

1725 ### 2.8.3.3 *Memory Protection*

1726 When an object is mapped, various application accesses to the mapped region may result in
1727 signals. In this context, SIGBUS is used to indicate an error using the mapped object, and
1728 SIGSEGV is used to indicate a protection violation or misuse of an address:

1729 - A mapping may be restricted to disallow some types of access.
1730 - Write attempts to memory that was mapped without write access, or any access to
1731 memory mapped PROT_NONE, shall result in a SIGSEGV signal.
1732 - References to unmapped addresses shall result in a SIGSEGV signal.
1733 - Reference to whole pages within the mapping, but beyond the current length of the object,
1734 shall result in a SIGBUS signal.
1735 - The size of the object is unaffected by access beyond the end of the object (even if a
1736 SIGBUS is not generated).

1737 ### 2.8.3.4 *Typed Memory Objects*

1738 TYM The functionality described in this section shall be provided on implementations that support
1739 the Typed Memory Objects option (and the rest of this section is not further shaded for this
1740 option).

1741 Implementations may support the Typed Memory Objects option independently of support for
1742 memory mapped files or shared memory objects. Typed memory objects are implementation-
1743 configurable named storage pools accessible from one or more processors in a system, each via
1744 one or more ports, such as backplane buses, LANs, I/O channels, and so on. Each valid
1745 combination of a storage pool and a port is identified through a name that is defined at system
1746 configuration time, in an implementation-defined manner; the name may be independent of the
1747 file system. Using this name, a typed memory object can be opened and mapped into process
1748 address space. For a given storage pool and port, it is necessary to support both dynamic
1749 allocation from the pool as well as mapping at an application-supplied offset within the pool;
1750 when dynamic allocation has been performed, subsequent deallocation must be supported.
1751 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

PS The functionality described in this section shall be provided on implementations that support 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 1003.1-200x 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).

The POSIX model treats a "process" as an aggregation of system resources, including one or more threads that may be scheduled by the operating system on the processor(s) it controls. Although a process has its own set of scheduling attributes, these have an indirect effect (if any) on the scheduling behavior of individual threads as described below.

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. Additionally, the scheduling parameters of a thread (but not its scheduling policy) may be changed by application execution of the *pthread_setschedprio*( ) function.

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.

The effect of the process scheduling attributes on individual threads in the process is dependent on the scheduling contention scope of the threads (see Section 2.9.4 (on page 52)):

- For threads with system scheduling contention scope, the process scheduling attributes shall have no effect on the scheduling attributes or behavior either of the thread or an underlying kernel scheduling entity dedicated to that thread.
- For threads with process scheduling contention scope, the process scheduling attributes shall have no effect on the scheduling attributes of the thread. However, any underlying kernel scheduling entity used by these threads shall at all times behave as specified by the scheduling attributes of the containing process, and this behavior may affect the scheduling behavior of the process contention scope threads. For example, a process contention scope thread with scheduling policy SCHED_FIFO and the system maximum priority *H* (the value returned by *sched_get_priority_max*(SCHED_FIFO)) in a process with scheduling policy SCHED_RR and system minimum priority *L* (the value returned by *sched_get_priority_min*(SCHED_RR)) shall be subject to timeslicing and to preemption by any thread with an effective priority higher than *L*.

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 1003.1-200x, **<sched.h>**:

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

SCHED_RR Round robin scheduling policy.

SS 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 priority has been modified by *pthread_setschedprio*( ) is a running thread or is runnable, the effect on its position in the thread list depends on the direction of the modification, as follows:

a. If the priority is raised, the thread becomes the tail of the thread list.

b. If the priority is unchanged, the thread does not change position in the thread list.

c. If the priority is lowered, the thread becomes the head of the thread list.

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

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

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

SCHED_RR

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

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

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

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

SCHED_SPORADIC

SS | TSP The functionality described in this section shall be provided on implementations that support 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 time parameters: the replenishment period and the available execution capacity. The replenishment period is given by the *sched_ss_repl_period* member of the **sched_param** structure. The available execution capacity is initialized to the value given by the *sched_ss_init_budget* member of the same parameter. The sporadic server policy is identical to the SCHED_FIFO policy with some additional conditions that cause the thread's assigned priority to be switched between the values specified by the *sched_priority* and *sched_ss_low_priority* members of the **sched_param** structure.

The priority assigned to a thread using the sporadic server scheduling policy is determined in the following manner: if the available execution capacity is greater than zero and the number of

pending replenishment operations is strictly less than *sched_ss_max_repl*, the thread is assigned the priority specified by *sched_priority*; otherwise, the assigned priority shall be *sched_ss_low_priority*. If the value of *sched_priority* is less than or equal to the value of *sched_ss_low_priority*, the results are undefined. When active, the thread shall belong to the thread list corresponding to its assigned priority level, according to the mentioned priority assignment. The modification of the available execution capacity and, consequently of the assigned priority, is done as follows:

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

2. Each time the thread is inserted at the tail of the list associated with *sched_priority*—because as a blocked thread it became runnable with priority *sched_priority* or because a replenishment operation was performed—the time at which this operation is done is posted as the *activation_time*.

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

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

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

6. Each time a replenishment operation is scheduled, the amount of execution capacity to be replenished, *replenish_amount*, is set equal to the execution time consumed by the thread since the *activation_time*. The replenishment is scheduled to occur at *activation_time* plus *sched_ss_repl_period*. If the scheduled time obtained is before the current time, the 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 Base Definitions volume of IEEE Std 1003.1-200x, Section 3.155, Execution Time.

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.

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.

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 SS 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 **<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:

Member Type	Member Name	Description
time_t	*tv_sec*	Seconds.
long	*tv_nsec*	Nanoseconds.

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:

Member Type	Member Name	Description
struct timespec	*it_interval*	Timer period.
struct timespec	*it_value*	Timer expiration.

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

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

Timer Event Notification Control Block

Per-process timers may be created that notify the process of timer expirations by queuing a realtime extended signal. The **sigevent** structure, defined in the Base Definitions volume of IEEE Std 1003.1-200x, **<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 1003.1-200x, **<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.

MON 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.

MON 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.

MON 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

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

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

CPT | TCT CPU-time clocks measure execution or CPU time, which is defined in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.117, CPU Time (Execution Time). The mechanism used to measure execution time is described in the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.9, Measurement of Execution Time.

CPT 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**() function call, it is interpreted as the identifier of the CPU-time clock associated with the process making the function call.

TCT 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**() function call, it is interpreted as the identifier of the CPU-time clock associated with the thread making the function call.

2.9 Threads

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 1003.1-200x, 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 1003.1-200x shall be thread-safe, except that the following functions[1] need not be thread-safe.

asctime()	*ftw*()	*getservbyport*()	*putc_unlocked*()
basename()	*getc_unlocked*()	*getservent*()	*putchar_unlocked*()
catgets()	*getchar_unlocked*()	*getutxent*()	*putenv*()
crypt()	*getdate*()	*getutxid*()	*pututxline*()
ctime()	*getenv*()	*getutxline*()	*rand*()
dbm_clearerr()	*getgrent*()	*gmtime*()	*readdir*()
dbm_close()	*getgrgid*()	*hcreate*()	*setenv*()
dbm_delete()	*getgrnam*()	*hdestroy*()	*setgrent*()
dbm_error()	*gethostent*()	*hsearch*()	*setkey*()
dbm_fetch()	*getlogin*()	*inet_ntoa*()	*setpwent*()
dbm_firstkey()	*getnetbyaddr*()	*l64a*()	*setutxent*()
dbm_nextkey()	*getnetbyname*()	*lgamma*()	*strerror*()
dbm_open()	*getnetent*()	*lgammaf*()	*strsignal*()
dbm_store()	*getopt*()	*lgammal*()	*strtok*()
dirname()	*getprotobyname*()	*localeconv*()	*system*()
dlerror()	*getprotobynumber*()	*localtime*()	*ttyname*()
drand48()	*getprotoent*()	*lrand48*()	*unsetenv*()
encrypt()	*getpwent*()	*mrand48*()	*wcstombs*()
endgrent()	*getpwnam*()	*nftw*()	*wctomb*()
endpwent()	*getpwuid*()	*nl_langinfo*()	
endutxent()	*getservbyname*()	*ptsname*()	

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

---

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

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 1003.1-200x 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”.

A problem can occur if a process terminates while one of its threads holds a mutex lock. Depending on the mutex type, it might be possible for another thread to unlock the mutex and recover the state of the mutex. However, it is difficult to perform this recovery reliably.

Robust mutexes provide a means to enable the implementation to notify other threads in the event of a process terminating while one of its threads holds a mutex lock. The next thread that acquires the mutex is notified about the termination by the return value [EOWNERDEAD] from the locking function. The notified thread can then attempt to recover the state protected by the mutex, and if successful mark the state protected by the mutex as consistent by a call to *pthread_mutex_consistent*( ). If the notified thread is unable to recover the state, it can declare the state as not recoverable by a call to *pthread_mutex_unlock*( ) without a prior call to *pthread_mutex_consistent*( ).

Whether or not the state protected by a mutex can be recovered is dependent solely on the

application using robust mutexes. The robust mutex support provided in the implementation provides notification only that a mutex owner has terminated while holding a lock, or that the state of the mutex is not recoverable.

2.9.4 Thread Scheduling

TPS The functionality described in this section shall be provided on implementations that support 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*.

TSP 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 policy and attributes of the process. 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 44). All threads with system scheduling contention scope, regardless of the processes in which they reside, compete for the processor according to their priorities. Threads with process scheduling contention scope compete only with other threads with process scheduling contention scope within their process.

For application threads with scheduling allocation domains of size greater than one, the rules
TSP 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.

TSP If _POSIX_THREAD_SPORADIC_SERVER is defined, the rules defined for SCHED_SPORADIC in Scheduling Policies 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
TSP 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 Cancellation

The thread cancellation 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 cancellation requests pending in a number of ways and to perform application-specific cleanup processing when the notice of cancellation is acted upon.

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

The thread cancellation 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 cancellation 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, cancellation 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 cancellation request. The thread may control cancellation 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 1003.1-200x, **<pthread.h>**), cancellation 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_CANCEL_ASYNCHRONOUS (as defined in **<pthread.h>**), new or pending cancellation requests may be acted upon at any time. When cancelability is enabled and the cancelability type is PTHREAD_CANCEL_DEFERRED (as defined in **<pthread.h>**), cancellation requests are held pending until a cancellation point (see below) is reached. If cancelability is disabled, the setting of the cancelability type has no immediate effect as all cancellation 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 *Cancellation Points*

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

accept( )	*nanosleep*( )	*select*( )
aio_suspend( )	*open*( )	*sem_timedwait*( )
clock_nanosleep( )	*openat*( )	*sem_wait*( )
close( )	*pause*( )	*send*( )
connect( )	*poll*( )	*sendmsg*( )
creat( )	*pread*( )	*sendto*( )
fcntl( )†	*pselect*( )	*sigsuspend*( )
fdatasync( )	*pthread_cond_timedwait*( )	*sigtimedwait*( )
fsync( )	*pthread_cond_wait*( )	*sigwait*( )
getmsg( )	*pthread_join*( )	*sigwaitinfo*( )
getpmsg( )	*pthread_testcancel*( )	*sleep*( )
lockf( )	*putmsg*( )	*system*( )
mq_receive( )	*putpmsg*( )	*tcdrain*( )
mq_send( )	*pwrite*( )	*wait*( )
mq_timedreceive( )	*read*( )	*waitid*( )
mq_timedsend( )	*readv*( )	*waitpid*( )
msgrcv( )	*recv*( )	*write*( )
msgsnd( )	*recvfrom*( )	*writev*( )
msync( )	*recvmsg*( )	

---

† When the *cmd* argument is F_SETLKW.

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

access()
asctime()
asctime_r()
catclose()
catgets()
catopen()
chmod()
chown()
closedir()
closelog()
ctermid()
ctime()
ctime_r()
dbm_close()
dbm_delete()
dbm_fetch()
dbm_nextkey()
dbm_open()
dbm_store()
dlclose()
dlopen()
dprintf()
endgrent()
endhostent()
endnetent()
endprotoent()
endpwent()
endservent()
endutxent()
faccessat()
fchmod()
fchmodat()
fchown()
fchownat()
fclose()
fcntl()†
fflush()
fgetc()
fgetpos()
fgets()
fgetwc()
fgetws()
fmtmsg()
fopen()
fpathconf()
fprintf()
fputc()
fputs()
fputwc()
fputws()
fread()
freopen()
fscanf()
fseek()
fseeko()
fsetpos()
fstat()
fstatat()
ftell()
ftello()
ftw()
futimesat()
fwprintf()
fwrite()
fwscanf()
getaddrinfo()
getc()
getc_unlocked()
getchar()
getchar_unlocked()
getcwd()
getdate()
getdelim()
getgrent()
getgrgid()
getgrgid_r()
getgrnam()
getgrnam_r()
gethostent()
gethostid()
gethostname()
getline()
getlogin()
getlogin_r()
getnameinfo()
getnetbyaddr()
getnetbyname()
getnetent()
getopt()††
getprotobyname()
getprotobynumber()
getprotoent()
getpwent()
getpwnam()
getpwnam_r()
getpwuid()
getpwuid_r()
gets()
getservbyname()
getservbyport()
getservent()
getutxent()
getutxid()
getutxline()
getwc()
getwchar()
glob()
iconv_close()
iconv_open()
ioctl()
link()
linkat()
lio_listio()
localtime()
localtime_r()
lseek()
lstat()
mkdir()
mkdirat()
mkdtemp()
mkfifo()
mkfifoat()
mknod()
mknodat()
mkstemp()
mktime()
nftw()
opendir()
openlog()
pathconf()
pclose()
perror()
popen()
posix_fadvise()
posix_fallocate()
posix_madvise()
posix_openpt()
posix_spawn()
posix_spawnp()
posix_trace_clear()
posix_trace_close()
posix_trace_create()
posix_trace_create_withlog()
posix_trace_eventtypelist_getnext_id()
posix_trace_eventtypelist_rewind()
posix_trace_flush()
posix_trace_get_attr()
posix_trace_get_filter()
posix_trace_get_status()
posix_trace_getnext_event()

posix_trace_open( )	*readlink*( )	*symlink*( )
posix_trace_rewind( )	*readlinkat*( )	*symlinkat*( )
posix_trace_set_filter( )	*remove*( )	*sync*( )
posix_trace_shutdown( )	*rename*( )	*syslog*( )
posix_trace_timedgetnext_event( )	*renameat*( )	*tmpfile*( )
posix_typed_mem_open( )	*rewind*( )	*tmpnam*( )
printf( )	*rewinddir*( )	*ttyname*( )
psiginfo( )	*scandir*( )	*ttyname_r*( )
psignal( )	*scanf*( )	*tzset*( )
pthread_rwlock_rdlock( )	*seekdir*( )	*ungetc*( )
pthread_rwlock_timedrdlock( )	*semop*( )	*ungetwc*( )
pthread_rwlock_timedwrlock( )	*setgrent*( )	*unlink*( )
pthread_rwlock_wrlock( )	*sethostent*( )	*unlinkat*( )
putc( )	*setnetent*( )	*utime*( )
putc_unlocked( )	*setprotoent*( )	*utimes*( )
putchar( )	*setpwent*( )	*vfprintf*( )
putchar_unlocked( )	*setservent*( )	*vfwprintf*( )
puts( )	*setutxent*( )	*vprintf*( )
pututxline( )	*sigpause*( )	*vwprintf*( )
putwc( )	*stat*( )	*wcsftime*( )
putwchar( )	*strerror*( )	*wordexp*( )
readdir( )	*strerror_r*( )	*wprintf*( )
readdir_r( )	*strftime*( )	*wscanf*( )

An implementation shall not introduce cancellation points into any other functions specified in this volume of IEEE Std 1003.1-200x.

The side effects of acting upon a cancellation 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 cancellation cleanup handlers are called.

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

2.9.5.3 *Thread Cancellation Cleanup Handlers*

Each thread maintains a list of cancellation 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 cancellation request is acted upon, or when a thread calls *pthread_exit*( ), the thread first disables cancellation by setting its cancelability state to PTHREAD_CANCEL_DISABLE and its cancelability type to PTHREAD_CANCEL_DEFERRED. The cancelability state shall remain set to PTHREAD_CANCEL_DISABLE until the thread has terminated. The behavior is undefined if

---

† For any value of the *cmd* argument.

†† If *opterr* is non-zero.

a cancellation cleanup handler or thread-specific data destructor routine changes the cancelability state to PTHREAD_CANCEL_ENABLE.

The routines in the thread's list of cancellation cleanup handlers 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). When the cancellation cleanup handler for a scope is invoked, the storage for that scope remains valid. If the last cancellation cleanup handler returns, thread-specific data destructors (if any) associated with thread-specific data keys for which the thread has non-NULL values will be run, in unspecified order, as described for *pthread_key_create*( ).

After all cancellation cleanup handlers and thread-specific data destructors have returned, thread execution is terminated. If the thread has terminated because of a call to *pthread_exit*( ), the *value_ptr* argument is made available to any threads joining with the target. If the thread has terminated by acting on a cancellation request, 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.

A side effect of acting upon a cancellation request while in a condition variable wait is that the mutex is re-acquired before calling the first cancellation 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 cancellation 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 1003.1-200x 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 following functions shall be atomic with respect to each other in the effects specified in IEEE Std 1003.1-200x when they operate on regular files or symbolic links:

chmod( )	*fchownat*( )	*linkat*( )	*readlink*( )	*unlinkat*( )
chown( )	*fcntl*( )	*lseek*( )	*readv*( )	*utime*( )
close( )	*fstat*( )	*lstat*( )	*rename*( )	*utimes*( )
creat( )	*fstatat*( )	*open*( )	*renameat*( )	*write*( )
dup2( )	*ftruncate*( )	*openat*( )	*stat*( )	*writev*( )
fchmod( )	*futimesat*( )	*pread*( )	*symlink*( )	
fchmodat( )	*lchown*( )	*read*( )	*truncate*( )	
fchown( )	*link*( )	*pwrite*( )	*unlink*( )	

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.

2.9.8 Use of Application-Managed Thread Stacks

An "application-managed thread stack" is a region of memory allocated by the application—for example, memory returned by the *malloc*( ) or *mmap*( ) functions—and designated as a stack through the act of passing the address and size of the stack, respectively, as the *stackaddr* and *stacksize* arguments to *pthread_attr_setstack*( ). Application-managed stacks allow the application to precisely control the placement and size of a stack.

The application grants to the implementation permanent ownership of and control over the application-managed stack when the attributes object in which the *stack* or *stackaddr* attribute has been set is used, either by presenting that attribute's object as the *attr* argument in a call to *pthread_create*( ) that completes successfully, or by storing a pointer to the attributes object in the *sigev_notify_attributes* member of a **struct sigevent** and passing that **struct sigevent** to a function accepting such argument that completes successfully. The application may thereafter utilize the memory within the stack only within the normal context of stack usage within or properly synchronized with a thread that has been scheduled by the implementation with stack pointer value(s) that are within the range of that stack. In particular, the region of memory cannot be freed, nor can it be later specified as the stack for another thread.

When specifying an attributes object with an application-managed stack through the *sigev_notify_attributes* member of a **struct sigevent**, the results are undefined if the requested signal is generated multiple times (as for a repeating timer).

Until an attributes object in which the *stack* or *stackaddr* attribute has been set is used, the application retains ownership of and control over the memory allocated to the stack. It may free or reuse the memory as long as it either deletes the attributes object, or before using the attributes object replaces the stack by making an additional call to *pthread_attr_setstack*( ), that was used originally to designate the stack. There is no mechanism to retract the reference to an application-managed stack by an existing attributes object.

Once an attributes object with an application-managed stack has been used, that attributes object cannot be used again by a subsequent call to *pthread_create*( ) or any function accepting a **struct sigevent** with *sigev_notify_attributes* containing a pointer to the attributes object, without designating an unused application-managed stack by making an additional call to *pthread_attr_setstack*( ).

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 61), and is associated with a specific protocol, detailed in Section 2.10.3 (on page 60). 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 67), Section 2.10.19 (on page 68), and Section 2.10.20 (on page 68), 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 1003.1-200x, **<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 (on page 61). 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 RS 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 that do not specify a peer address 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.

RS 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 67), Section 2.10.19 (on page 68), and Section 2.10.20 (on page 68). 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, *getsockopt*( ) and *setsockopt*( ). 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 2-1 Value of Level for Socket Options

Name	Description
SOL_SOCKET	Options are intended for the sockets level.

Table 2-2 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.

Table 2-2 Socket-Level Options

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

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

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

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

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

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

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

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

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

The 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 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 65), the option value is a **timeval** structure with the number of seconds and microseconds specifying the limit on how long to wait for an output operation to complete. If a send operation has blocked for this much time, it shall return with a partial count or *errno* set to [EWOULDBLOCK] if no data were sent. The default for this option is the value zero, which indicates that a send operation will not time out. It is implementation-defined whether the SO_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 1003.1-200x, 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 1003.1-200x, Chapter 13, Headers) and shall be aligned at an appropriate boundary so that pointers to it can be cast as pointers to **sockaddr_un** structures and used to access the fields of those structures without alignment problems. When a **sockaddr_storage** structure is cast as a **sockaddr_un** structure, the *ss_family* field maps onto the *sun_family* field.

2.10.18 Use of Sockets over Internet Protocols

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

SOCK_STREAM	IPPROTO_TCP.
SOCK_DGRAM	IPPROTO_UDP.
RS SOCK_RAW	IPPROTO_RAW.

SOCK_SEQPACKET Unspecified.

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

2.10.19 Use of Sockets over Internet Protocols Based on IPv4

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

2.10.19.1 Headers

The symbolic constant AF_INET defined in the **<sys/socket.h>** header is used to identify the IPv4 Internet address family. The **<netinet/in.h>** header contains other definitions used in connection with IPv4 Internet sockets. See the Base Definitions volume of IEEE Std 1003.1-200x, 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 1003.1-200x, 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

IP6 This section describes extensions to support sockets over Internet protocols based on IPv6. The functionality described in this section shall be provided on implementations that support 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 69).

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 (on page 69).

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.

2854 *2.10.20.3 Interface Identification*

2855 Each local interface is assigned a unique positive integer as a numeric index. Indexes start at 1;
2856 zero is not used. There may be gaps so that there is no current interface for a particular positive
2857 index. Each interface also has a unique implementation-defined name.

2858 *2.10.20.4 Options*

2859 The following options apply at the IPPROTO_IPV6 level:

2860 IPV6_JOIN_GROUP
2861 When set via *setsockopt*( ), it joins the application to a multicast group on an interface
2862 (identified by its index) and addressed by a given multicast address, enabling packets sent
2863 to that address to be read via the socket. If the interface index is specified as zero, the
2864 system selects the interface (for example, by looking up the address in a routing table and
2865 using the resulting interface).

2866 An attempt to read this option using *getsockopt*( ) shall result in an [EOPNOTSUPP] error.

2867 The parameter type of this option is a pointer to an **ipv6_mreq** structure.

2868 IPV6_LEAVE_GROUP
2869 When set via *setsockopt*( ), it removes the application from the multicast group on an
2870 interface (identified by its index) and addressed by a given multicast address.

2871 An attempt to read this option using *getsockopt*( ) shall result in an [EOPNOTSUPP] error.

2872 The parameter type of this option is a pointer to an **ipv6_mreq** structure.

2873 IPV6_MULTICAST_HOPS
2874 The value of this option is the hop limit for outgoing multicast IPv6 packets sent via the
2875 socket. Its possible values are the same as those of IPV6_UNICAST_HOPS. If the
2876 IPV6_MULTICAST_HOPS option is not set, a value of 1 is assumed. This option can be set
2877 via *setsockopt*( ) and read via *getsockopt*( ).

2878 The parameter type of this option is a pointer to an **int**. (Default value: 1)

2879 IPV6_MULTICAST_IF
2880 The index of the interface to be used for outgoing multicast packets. It can be set via
2881 *setsockopt*( ) and read via *getsockopt*( ). If the interface index is specified as zero, the system
2882 selects the interface (for example, by looking up the address in a routing table and using the
2883 resulting interface).

2884 The parameter type of this option is a pointer to an **unsigned int**. (Default value: 0)

2885 IPV6_MULTICAST_LOOP
2886 This option controls whether outgoing multicast packets should be delivered back to the
2887 local application when the sending interface is itself a member of the destination multicast
2888 group. If it is set to 1 they are delivered. If it is set to 0 they are not. Other values result in an
2889 [EINVAL] error. This option can be set via *setsockopt*( ) and read via *getsockopt*( ).

2890 The parameter type of this option is a pointer to an **unsigned int** which is used as a Boolean
2891 value. (Default value: 1)

2892 IPV6_UNICAST_HOPS
2893 The value of this option is the hop limit for outgoing unicast IPv6 packets sent via the
2894 socket. If the option is not set, or is set to –1, the system selects a default value. Attempts to
2895 set a value less than –1 or greater than 255 shall result in an [EINVAL] error. This option can
2896 be set via *setsockopt*( ) and read via *getsockopt*( ).

2897 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 applications, the IPv6_V6ONLY socket option is defined. When this option is turned on, the socket can be used to send and receive IPv6 packets only. This is an IPPROTO_IPV6-level option.

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

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

2.10.20.5 Headers

The symbolic constant AF_INET6 is defined in the **<sys/socket.h>** header to identify the IPv6 Internet address family. See the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers.

The **sockaddr_storage** structure defined in **<sys/socket.h>** shall be large enough to accommodate a **sockaddr_in6** structure (see the **<netinet/in.h>** header defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers) and shall be aligned at an appropriate boundary so that pointers to it can be cast as pointers to **sockaddr_in6** structures and used to access the fields of those structures without alignment problems. When a **sockaddr_storage** structure is cast as a **sockaddr_in6** structure, the *ss_family* field maps onto the *sin6_family* field.

The **<netinet/in.h>**, **<arpa/inet.h>**, and **<netdb.h>** headers contain other definitions used in connection with IPv6 Internet sockets; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers.

2.11 Tracing

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

The tracing facilities defined in IEEE Std 1003.1-200x allow a process to select a set of trace event types, to activate a trace stream of the selected trace events as they occur in the flow of execution, and to retrieve the recorded trace events.

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

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

These operations can be done for an active trace stream. The analyzer process retrieves the

traced events either at runtime, when the trace stream has not yet been shut down, but is still recording trace events; or after opening a trace log that had been previously recorded and shut down. These three logically different operations can be performed by the same process, or can be distributed into different processes.

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

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

The events to be traced belong to two classes:

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

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

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

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

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

Results are undefined if the application performs any operation on a file descriptor associated with an active or pre-recorded trace stream until *posix_trace_shutdown*() or *posix_trace_close*() is called for that trace stream. Results are also undefined if the analyzer process and the traced process do not share the same programming environment (see *c99*, Programming Environments in the Shell and Utilities volume of IEEE Std 1003.1-200x.

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

2.11.1 Tracing Data Definitions

2.11.1.1 *Structures*

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

posix_trace_status_info Structure

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

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

Member Type	Member Name	Description
int	*posix_stream_status*	The operating mode of the trace stream.
int	*posix_stream_full_status*	The full status of the trace stream.
int	*posix_stream_overrun_status*	Indicates whether trace events were lost in the trace stream.

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

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

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

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

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

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

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

POSIX_TRACE_NOT_FULL
: There is still space available in the trace stream.

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

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

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

3065 POSIX_TRACE_NOT_FLUSHING
3066 No flush operation is in progress.

3067 The *posix_stream_flush_status* member shall be set to POSIX_TRACE_FLUSHING if a flush
3068 operation is in progress either due to a call to the *posix_trace_flush*( ) function (explicit or caused
3069 by a trace stream shutdown operation) or because the trace stream has become full with the
3070 *stream-full-policy* attribute set to POSIX_TRACE_FLUSH. The *posix_stream_flush_status* member
3071 shall be set to POSIX_TRACE_NOT_FLUSHING if no flush operation is in progress.

3072 The *posix_stream_flush_error* member shall be set to zero if no error occurred during flushing. If
3073 an error occurred during a previous flushing operation, the *posix_stream_flush_error* member
3074 shall be set to the value of the first error that occurred. If more than one error occurs while
3075 flushing, error values after the first shall be discarded. The *posix_stream_flush_error* member is
3076 reset to zero after its value is read.

3077 The *posix_log_overrun_status* member indicates whether trace events were lost in the trace log,
3078 and shall have one of the following values defined by manifest constants in the **<trace.h>**
3079 header:

3080 POSIX_TRACE_OVERRUN
3081 At least one trace event was lost.

3082 POSIX_TRACE_NO_OVERRUN
3083 No trace events were lost.

3084 When the corresponding trace stream is created, the *posix_log_overrun_status* member shall be set
3085 to POSIX_TRACE_NO_OVERRUN. Whenever an overrun occurs, this status shall be set to
3086 POSIX_TRACE_OVERRUN. The *posix_log_overrun_status* member is reset to zero after its value
3087 is read.

3088 The *posix_log_full_status* member indicates the full status of the trace log, and it shall have one of
3089 the following values defined by manifest constants in the **<trace.h>** header:

3090 POSIX_TRACE_FULL
3091 The space in the trace log is exhausted.

3092 POSIX_TRACE_NOT_FULL
3093 There is still space available in the trace log.

3094 The *posix_log_full_status* member is only meaningful if the *log-full-policy* attribute is either
3095 POSIX_TRACE_UNTIL_FULL or POSIX_TRACE_LOOP.

3096 For an active trace stream without log, that is created by the *posix_trace_create*( ) function, the
3097 *posix_log_overrun_status* member shall be set to POSIX_TRACE_NO_OVERRUN and the
3098 *posix_log_full_status* member shall be set to POSIX_TRACE_NOT_FULL.

3099 **posix_trace_event_info Structure**

3100 The trace event structure **posix_trace_event_info** contains the information for one recorded
3101 trace event. This structure is returned by the set of functions *posix_trace_getnext_event*( ),
3102 *posix_trace_timedgetnext_event*( ), and *posix_trace_trygetnext_event*( ).

3103 The **posix_trace_event_info** structure defined in **<trace.h>** shall contain at least the following
3104 members:

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

In addition, the **posix_trace_event_info** structure defined in **<trace.h>** shall contain the following additional member:

Member Type	Member Name	Description
pthread_t	*posix_thread_id*	Thread ID of the thread that generated the trace event.

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.

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.

3151 #### 2.11.1.2 Trace Stream Attributes

3152 Trace streams have attributes that compose the **posix_trace_attr_t** trace stream attributes object.
3153 This object shall contain at least the following attributes:

3154 - The *generation-version* attribute identifies the origin and version of the trace system.
3155 - The *trace-name* attribute is a character string defined by the trace controller, and that
3156 identifies the trace stream.
3157 - The *creation-time* attribute represents the time of the creation of the trace stream.
3158 - The *clock-resolution* attribute defines the clock resolution of the clock used to generate
3159 timestamps.
3160 - The *stream-min-size* attribute defines the minimum size in bytes of the trace stream strictly
3161 reserved for the trace events.
3162 - The *stream-full-policy* attribute defines the policy followed when the trace stream is full; its
3163 value is POSIX_TRACE_LOOP, POSIX_TRACE_UNTIL_FULL, or POSIX_TRACE_FLUSH.
3164 - The *max-data-size* attribute defines the maximum record size in bytes of a trace event.

3165 In addition, if the Trace option and the Trace Inherit option are both supported, the
3166 **posix_trace_attr_t** trace stream creation attributes object shall contain at least the following
3167 attributes:

3168 - The *inheritance* attribute specifies whether a newly created trace stream will inherit tracing
3169 in its parent's process trace stream. It is either POSIX_TRACE_INHERITED or
3170 POSIX_TRACE_CLOSE_FOR_CHILD.

3171 In addition, if the Trace option and the Trace Log option are both supported, the
3172 **posix_trace_attr_t** trace stream creation attributes object shall contain at least the following
3173 attribute:

3174 - If the file type corresponding to the trace log supports the POSIX_TRACE_LOOP or the
3175 POSIX_TRACE_UNTIL_FULL policies, the *log-max-size* attribute defines the maximum
3176 size in bytes of the trace log associated with an active trace stream. Other stream data—for
3177 example, trace attribute values—shall not be included in this size.
3178 - The *log-full-policy* attribute defines the policy of a trace log associated with an active trace
3179 stream to be POSIX_TRACE_LOOP, POSIX_TRACE_UNTIL_FULL, or
3180 POSIX_TRACE_APPEND.

3181 ### 2.11.2 Trace Event Type Definitions

3182 #### 2.11.2.1 System Trace Event Type Definitions

3183 The following system trace event types, defined in the **<trace.h>** header, track the invocation of
3184 the trace operations:

3185 - POSIX_TRACE_START shall be associated with a trace start operation.
3186 - POSIX_TRACE_STOP shall be associated with a trace stop operation.
3187 - If the Trace Event Filter option is supported, POSIX_TRACE_FILTER shall be associated
3188 with a trace event type filter change operation.

3189 The following system trace event types, defined in the **<trace.h>** header, report operational trace
3190 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 2-3 Trace Option: System Trace Events

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

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

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

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

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

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

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

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

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

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

2.11.2.2 *User Trace Event Type Definitions*

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

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

Table 2-7 Trace Option: User Trace Event

Event Name	Constant
posix_trace_unnamed_userevent	POSIX_TRACE_UNNAMED_USEREVENT

2.11.3 Trace Functions

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

- A trace controller process requires functions to set up and customize all the resources needed to run a trace stream, including:
  - Attribute initialization and destruction (*posix_trace_attr_init*())
  - Identification information manipulation (*posix_trace_attr_getgenversion*())
  - Trace system behavior modification (*posix_trace_attr_getinherited*())
  - Trace stream and trace log size set (*posix_trace_attr_getmaxusereventsize*())
  - Trace stream creation, flush, and shutdown (*posix_trace_create*())
  - Trace stream and trace log clear (*posix_trace_clear*())
  - Trace event type identifier manipulation (*posix_trace_trid_eventid_open*())
  - Trace event type identifier list exploration (*posix_trace_eventtypelist_getnext_id*())
  - Trace event type set manipulation (*posix_trace_eventset_empty*())
  - Trace event type filter set (*posix_trace_set_filter*())
  - Trace stream start and stop (*posix_trace_start*())
  - Trace stream information and status read (*posix_trace_get_attr*())
- A traced process requires functions to instrument trace points:
  - Trace event type identifiers definition and trace points insertion (*posix_trace_event*())
- A trace analyzer process requires functions to retrieve information from a trace stream and trace log:
  - Identification information read (*posix_trace_attr_getgenversion*())
  - Trace system behavior information read (*posix_trace_attr_getinherited*())
  - Trace stream and trace log size get (*posix_trace_attr_getmaxusereventsize*())
  - Trace event type identifier manipulation (*posix_trace_trid_eventid_open*())
  - Trace event type identifier list exploration (*posix_trace_eventtypelist_getnext_id*())
  - Trace log open, rewind, and close (*posix_trace_open*())
  - Trace stream information and status read (*posix_trace_get_attr*())

— Trace event read (*posix_trace_getnext_event*())

2.12 Data Types

2.12.1 Defined 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.

Defined Type	Description
cc_t	Type used for terminal special characters.
clock_t	Integer or real-floating type used for processor times, as defined in the ISO C standard.
clockid_t	Used for clock ID type in some timer functions.
dev_t	Arithmetic type used for device numbers.
DIR	Type representing a directory stream.
div_t	Structure type returned by the *div*() function.
FILE	Structure containing information about a file.
glob_t	Structure type used in pathname pattern matching.
fpos_t	Type containing all information needed to specify uniquely every position within a file.
gid_t	Integer type used for group IDs.
iconv_t	Type used for conversion descriptors.
id_t	Integer type used as a general identifier; can be used to contain at least the largest of a **pid_t**, **uid_t**, or **gid_t**.
ino_t	Unsigned integer type used for file serial numbers.
key_t	Arithmetic type used for XSI interprocess communication.
ldiv_t	Structure type returned by the *ldiv*() function.
mode_t	Integer type used for file attributes.
mqd_t	Used for message queue descriptors.
nfds_t	Integer type used for the number of file descriptors.
nlink_t	Integer type used for link counts.
off_t	Signed integer type used for file sizes.
pid_t	Signed integer type used for process and process group IDs.
pthread_attr_t	Used to identify a thread attribute object.
pthread_cond_t	Used for condition variables.
pthread_condattr_t	Used to identify a condition attribute object.
pthread_key_t	Used for thread-specific data keys.
pthread_mutex_t	Used for mutexes.
pthread_mutexattr_t	Used to identify a mutex attribute object.
pthread_once_t	Used for dynamic package initialization.
pthread_rwlock_t	Used for read-write locks.
pthread_rwlockattr_t	Used for read-write lock attributes.
pthread_t	Used to identify a thread.
ptrdiff_t	Signed integer type of the result of subtracting two pointers.
regex_t	Structure type used in regular expression matching.
regmatch_t	Structure type used in regular expression matching.
rlim_t	Unsigned integer type used for limit values, to which objects of

Defined Type	Description
	type **int** and **off_t** can be cast without loss of value.
sem_t	Type used in performing semaphore operations.
sig_atomic_t	Integer type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts.
sigset_t	Integer or structure type of an object used to represent sets of signals.
size_t	Unsigned integer type used for size of objects.
speed_t	Type used for terminal baud rates.
ssize_t	Signed integer type used for a count of bytes or an error indication.
suseconds_t	Signed integer type used for time in microseconds.
tcflag_t	Type used for terminal modes.
time_t	Integer or real-floating type used for time in seconds, as defined in the ISO C standard.
timer_t	Used for timer ID returned by the *timer_create*( ) function.
uid_t	Integer type used for user IDs.
va_list	Type used for traversing variable argument lists.
wchar_t	Integer type whose range of values can represent distinct codes for all members of the largest extended character set specified by the supported locales.
wctype_t	Scalar type which represents a character class descriptor.
wint_t	Integer type capable of storing any valid value of **wchar_t** or WEOF.
wordexp_t	Structure type used in word expansion.

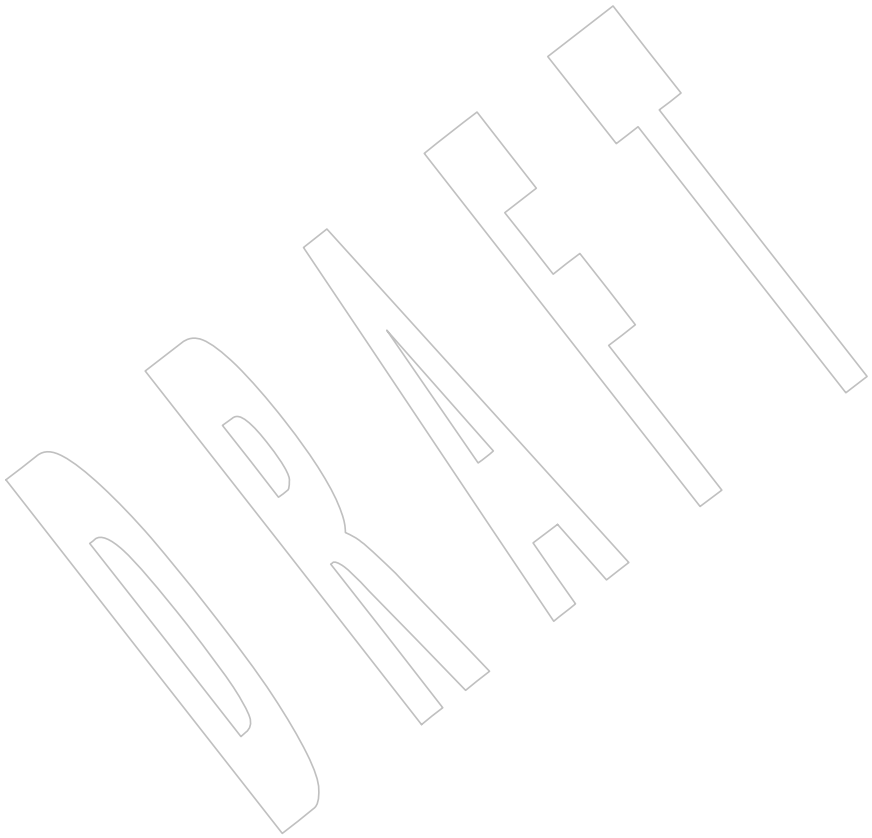
2.12.2 The char Type

The type **char** is defined as a single byte; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 3, Definitions (Byte and Character).

2.12.3 Pointer Types

Conforming implementations shall support conversion of pointers of any type to **void** * and back without loss of information.

Note: The ISO C standard does not require this, but it is required for POSIX conformance.



3402 *Chapter 3*

3403 # *System Interfaces*

3404 This chapter describes the functions, macros, and external variables to support applications
3405 portability at the C-language source level.

NAME

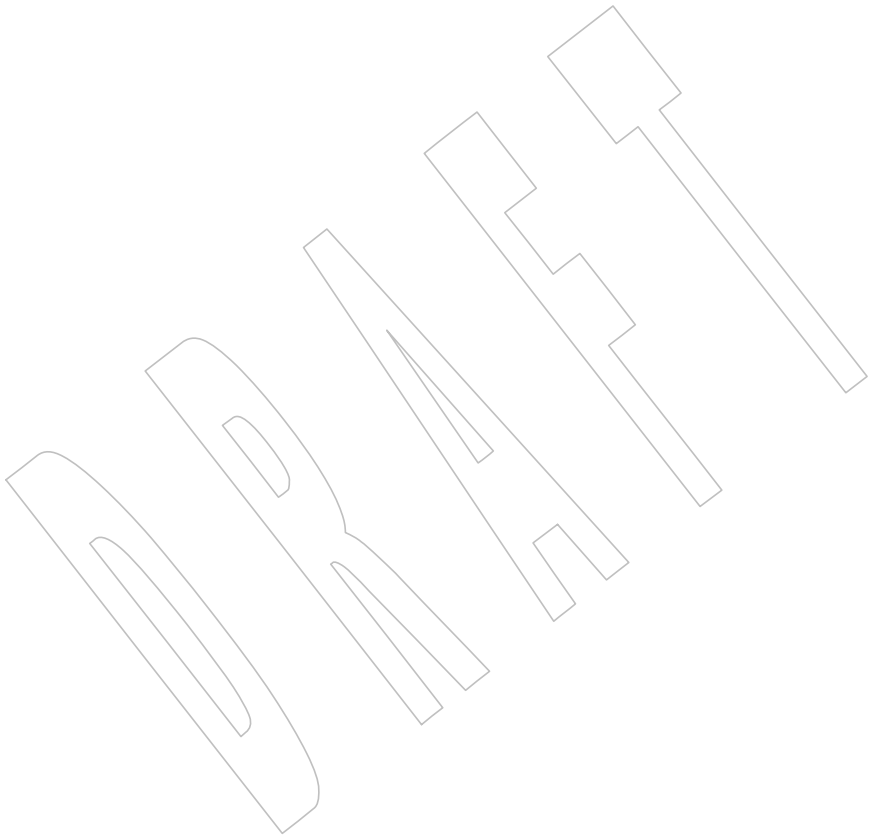
FD_CLR — macros for synchronous I/O multiplexing

SYNOPSIS

```
#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 <stdlib.h>

void _Exit(int status);

#include <unistd.h>

void _exit(int status);
```

DESCRIPTION

CX For *_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 1003.1-200x defers to the ISO C standard.

CX The value of *status* may be 0, EXIT_SUCCESS, EXIT_FAILURE, or any other value, though only the least significant 8 bits (that is, *status* & 0377) shall be available to a waiting parent process.

CX The *_Exit*( ) and *_exit*( ) functions shall be functionally equivalent.

CX The *_Exit*( ) and *_exit*( ) functions shall not call functions registered with *atexit*( ) nor any
CX registered signal handlers. Open streams shall not be flushed. Whether open streams are closed (without flushing) is implementation-defined. Finally, the calling process is terminated with the consequences described below.

Consequences of Process Termination

CX 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, functionality relating to the XSI option is shaded.

- All of the file descriptors, directory streams, conversion descriptors, and message catalog descriptors open in the calling process shall be closed.

XSI
- If the parent process of the calling process is executing a *wait*( ) or *waitpid*( ), and has neither set its SA_NOCLDWAIT flag nor set SIGCHLD to SIG_IGN, it shall be notified of termination of the calling process and the low-order eight bits (that is, bits 0377) of *status* shall be made available to it. If the parent is not waiting, the child's status shall be made available to it when the parent subsequently executes *wait*( ) or *waitpid*( ).

  The semantics of the *waitid*( ) function shall be equivalent to *wait*( ).

XSI
- If the parent process of the calling process is not executing a *wait*( ) or *waitpid*( ), and has neither set its SA_NOCLDWAIT flag nor set SIGCHLD to SIG_IGN, the calling process shall be transformed into a *zombie process*. A *zombie process* is an inactive process and it shall be deleted at some later time when its parent process executes *wait*( ) or *waitpid*( ).

XSI The semantics of the *waitid*( ) function shall be equivalent to *wait*( ).

- Termination of a process does not directly terminate its children. The sending of a SIGHUP signal as described below indirectly terminates children in some circumstances.

- Either:

  If the implementation supports the SIGCHLD signal, a SIGCHLD shall be sent to the parent process.

  Or:

XSI If the parent process has set its SA_NOCLDWAIT flag, or set SIGCHLD to SIG_IGN, the status shall be discarded, and the lifetime of the calling process shall end immediately. If SA_NOCLDWAIT is set, it is implementation-defined whether a SIGCHLD signal is sent to the parent process.

- The parent process ID of all of the existing child processes and zombie processes of the calling process 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.

XSI
- 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.

XSI
- 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*( ).

ML
- 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.

TYM
- Any blocks of typed memory that were mapped in the calling process shall be unmapped, as if *munmap*( ) was implicitly called to unmap them.

MSG
- 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 cancellation cleanup handlers or per-thread data destructors.

OB TRC
- 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 mapping of trace event names to trace event type identifiers of any process built for these trace streams may be deallocated.

3502 **RETURN VALUE**

3503 These functions do not return.

3504 **ERRORS**

3505 No errors are defined.

3506 **EXAMPLES**

3507 None.

3508 **APPLICATION USAGE**

3509 Normally applications should use *exit*( ) rather than *_Exit*( ) or *_exit*( ).

3510 **RATIONALE**

3511 **Process Termination**

3512 Early proposals drew a distinction between normal and abnormal process termination.
3513 Abnormal termination was caused only by certain signals and resulted in implementation-
3514 defined "actions", as discussed below. Subsequent proposals distinguished three types of
3515 termination: *normal termination* (as in the current specification), *simple abnormal termination*, and
3516 *abnormal termination with actions*. Again the distinction between the two types of abnormal
3517 termination was that they were caused by different signals and that implementation-defined
3518 actions would result in the latter case. Given that these actions were completely implementation-
3519 defined, the early proposals were only saying when the actions could occur and how their
3520 occurrence could be detected, but not what they were. This was of little or no use to conforming
3521 applications, and thus the distinction is not made in this volume of IEEE Std 1003.1-200x.

3522 The implementation-defined actions usually include, in most historical implementations, the
3523 creation of a file named **core** in the current working directory of the process. This file contains an
3524 image of the memory of the process, together with descriptive information about the process,
3525 perhaps sufficient to reconstruct the state of the process at the receipt of the signal.

3526 There is a potential security problem in creating a **core** file if the process was set-user-ID and the
3527 current user is not the owner of the program, if the process was set-group-ID and none of the
3528 user's groups match the group of the program, or if the user does not have permission to write
3529 in the current directory. In this situation, an implementation either should not create a **core** file
3530 or should make it unreadable by the user.

3531 Despite the silence of this volume of IEEE Std 1003.1-200x on this feature, applications are
3532 advised not to create files named **core** because of potential conflicts in many implementations.
3533 Some implementations use a name other than **core** for the file; for example, by appending the
3534 process ID to the filename.

3535 **Terminating a Process**

3536 It is important that the consequences of process termination as described occur regardless of
3537 whether the process called *_exit*( ) (perhaps indirectly through *exit*( )) or instead was terminated
3538 due to a signal or for some other reason. Note that in the specific case of *exit*( ) this means that
3539 the *status* argument to *exit*( ) is treated in the same way as the *status* argument to *_exit*( ).

3540 A language other than C may have other termination primitives than the C-language *exit*( )
3541 function, and programs written in such a language should use its native termination primitives,
3542 but those should have as part of their function the behavior of *_exit*( ) as described.
3543 Implementations in languages other than C are outside the scope of this version of this volume
3544 of IEEE Std 1003.1-200x, however.

3545 As required by the ISO C standard, using **return** from *main*( ) has the same behavior (other than
3546 with respect to language scope issues) as calling *exit*( ) with the returned value. Reaching the end
3547 of the *main*( ) function has the same behavior as calling *exit*(0).

A value of zero (or EXIT_SUCCESS, which is required to be zero) for the argument *status* conventionally indicates successful termination. This corresponds to the specification for *exit*( ) in the ISO C standard. The convention is followed by utilities such as *make* and various shells, which interpret a zero status from a child process as success. For this reason, applications should not call *exit*(0) or *_exit*(0) when they terminate unsuccessfully; for example, in signal-catching functions.

Historically, the implementation-defined process that inherits children whose parents have terminated without waiting on them is called *init* and has a process ID of 1.

The sending of a SIGHUP to the foreground process group when a controlling process terminates corresponds to somewhat different historical implementations. In System V, the kernel sends a SIGHUP on termination of (essentially) a controlling process. In 4.2 BSD, the kernel does not send SIGHUP in a case like this, but the termination of a controlling process is usually noticed by a system daemon, which arranges to send a SIGHUP to the foreground process group with the *vhangup*( ) function. However, in 4.2 BSD, due to the behavior of the shells that support job control, the controlling process is usually a shell with no other processes in its process group. Thus, a change to make *_exit*( ) behave this way in such systems should not cause problems with existing applications.

The termination of a process may cause a process group to become orphaned in either of two ways. The connection of a process group to its parent(s) outside of the group depends on both the parents and their children. Thus, a process group may be orphaned by the termination of the last connecting parent process outside of the group or by the termination of the last direct descendant of the parent process(es). In either case, if the termination of a process causes a process group to become orphaned, processes within the group are disconnected from their job control shell, which no longer has any information on the existence of the process group. Stopped processes within the group would languish forever. In order to avoid this problem, newly orphaned process groups that contain stopped processes are sent a SIGHUP signal and a SIGCONT signal to indicate that they have been disconnected from their session. The SIGHUP signal causes the process group members to terminate unless they are catching or ignoring SIGHUP. Under most circumstances, all of the members of the process group are stopped if any of them are stopped.

The action of sending a SIGHUP and a SIGCONT signal to members of a newly orphaned process group is similar to the action of 4.2 BSD, which sends SIGHUP and SIGCONT to each stopped child of an exiting process. If such children exit in response to the SIGHUP, any additional descendants receive similar treatment at that time. In this volume of IEEE Std 1003.1-200x, the signals are sent to the entire process group at the same time. Also, in this volume of IEEE Std 1003.1-200x, 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 1003.1-200x 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 1003.1-200x, this is equivalent to the *_exit*( ) function.

FUTURE DIRECTIONS

None.

SEE ALSO

atexit( ), *exit( )*, *mlock( )*, *mlockall( )*, *mq_close( )*, *munmap( )*, *posix_trace_shutdown( )*, *sem_close( )*, *semop( )*, *setpgid( )*, *setsid( )*, *shmget( )*, *wait( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/16 is applied, correcting grammar in the DESCRIPTION.

Issue 7

Austin Group Interpretation 1003.1-2001 #031 is applied, separating these functions from the *exit( )* function.

Austin Group Interpretation 1003.1-2001 #085 is applied, clarifying the text regarding flushing of streams and closing of temporary files.

Functionality relating to the Asynchronous Input and Output, Memory Mapped Files, and Semaphores options is moved to the Base.

NAME

_longjmp, _setjmp — non-local goto

SYNOPSIS

OB XSI

```
#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 *sigsetjmp*( ), 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 *sigsetjmp*( ) respectively.

RATIONALE

None.

FUTURE DIRECTIONS

The *_longjmp*( ) and *_setjmp*( ) functions may be removed in a future version.

SEE ALSO

longjmp( ), *setjmp*( ), *siglongjmp*( ), *sigsetjmp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<setjmp.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

3667 **Issue 5**

Moved from X/OPEN UNIX extension to BASE.

Issue 7

The *_longjmp*( ) and *_setjmp*( ) functions are marked obsolescent.

NAME

_tolower — transliterate uppercase characters to lowercase

SYNOPSIS

OB XSI
```
#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

Applications should use the *tolower*( ) function instead of the obsolescent *_tolower*( ) function.

RATIONALE

None.

FUTURE DIRECTIONS

The *_tolower*( ) function may be removed in a future version.

SEE ALSO

tolower( ), *isupper*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale, **<ctype.h>**

CHANGE HISTORY

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

Issue 6

The normative text is updated to avoid use of the term "must" for application requirements.

Issue 7

The *_tolower*( ) function is marked obsolescent.

NAME

_toupper — transliterate lowercase characters to uppercase

SYNOPSIS

OB 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

Applications should use the *toupper*( ) function instead of the obsolescent *_toupper*( ) function.

RATIONALE

None.

FUTURE DIRECTIONS

The *_toupper*( ) function may be removed in a future version.

SEE ALSO

islower( ), *toupper*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale, **<ctype.h>**

CHANGE HISTORY

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

Issue 6

The normative text is updated to avoid use of the term "must" for application requirements.

Issue 7

The *_toupper*( ) function is marked obsolescent.

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 thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

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

strtoul( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x, *uuencode*

CHANGE HISTORY

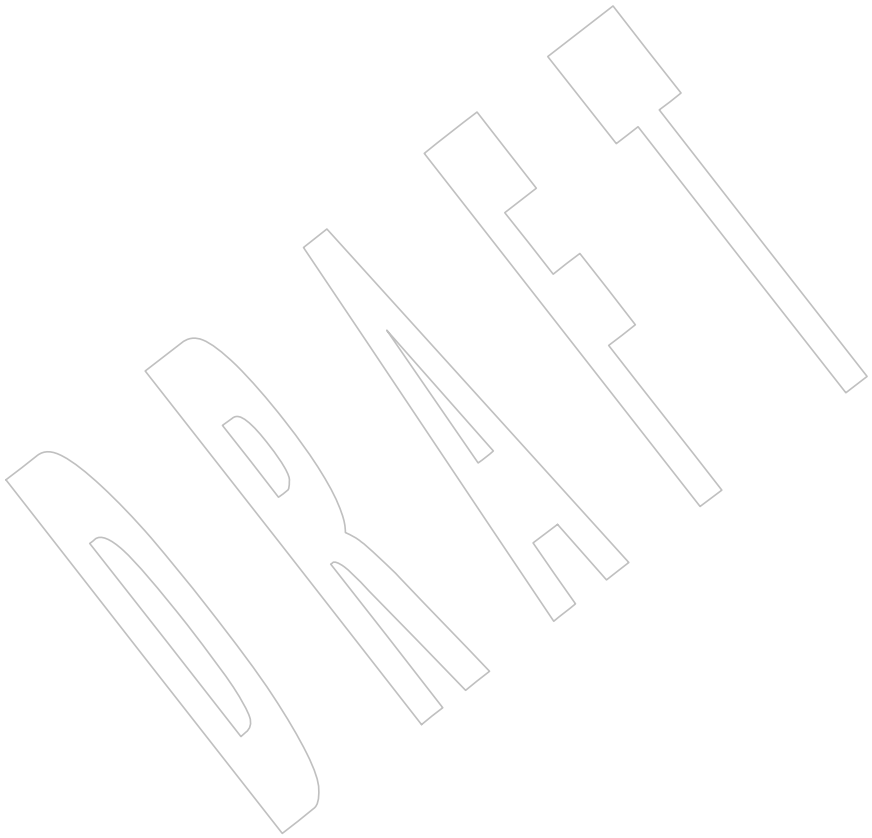
First released in Issue 4, Version 2.

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 the *l64a( )* function need not be reentrant is added to the DESCRIPTION.



NAME

abort — generate an abnormal process abort

SYNOPSIS

```
#include <stdlib.h>

void abort(void);
```

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 1003.1-200x 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.

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

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

CX 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-supplied functions.

RATIONALE

The ISO/IEC 9899:1999 standard requires the *abort*( ) function to be async-signal-safe. Since IEEE Std 1003.1-200x defers to the ISO C standard, this required a change to the DESCRIPTION from "shall include the effect of *fclose*( )" to "may include an attempt to effect *fclose*( )."

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

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

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

There were several reasons to remove this paragraph:

- No special processing of open message catalogs needs to be performed prior to abnormal process termination.
- The main reason to specifically mention that *abort*( ) includes the effect of *fclose*( ) on open streams is to flush output queued on the stream. Message catalogs in this context are read-only and, therefore, do not need to be flushed.

- The effect of *fclose*( ) on a message catalog descriptor is unspecified. Message catalog descriptors are allowed, but not required to be implemented using a file descriptor, but there is no mention in IEEE Std 1003.1-200x of a message catalog descriptor using a standard I/O stream FILE object as would be expected by *fclose*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

exit( ), *kill*( ), *raise*( ), *signal*( ), *wait*( ), *waitpid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

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

The Open Group Base Resolution bwg2002-003 is applied.

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

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

NAME

abs — return an integer absolute value

SYNOPSIS

```
#include <stdlib.h>

int abs(int i);
```

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 1003.1-200x 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 1003.1-200x, **<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*( ).

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

address_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.

[EINTR] The *accept*( ) function was interrupted by a signal that was caught before a valid connection arrived.

[EINVAL] The *socket* is not accepting connections.

[EMFILE] All file descriptors available to the process are currently open.

[ENFILE] The maximum number of file descriptors in the system are already open.

[ENOBUFS] No buffer space is available.

[ENOMEM] There was insufficient memory available to complete the operation.

[ENOTSOCK] The *socket* argument does not refer to a socket.

[EOPNOTSUPP] The socket type of the specified socket does not support accepting connections.

The *accept*( ) function may fail if:

OB XSR [EPROTO] A protocol error has occurred; for example, the STREAMS protocol stack has not been initialized.

EXAMPLES

None.

APPLICATION USAGE

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

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

bind( ), *connect*( ), *listen*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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 *accept*( ) prototype for alignment with the ISO/IEC 9899: 1999 standard.

Issue 7

SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

Austin Group Interpretation 1003.1-2001 #044 is applied, changing the "may fail" [ENOBUFS] and [ENOMEM] errors to become "shall fail" errors.

Functionality relating to XSI STREAMS is marked obsolescent.

NAME

access, faccessat — determine accessibility of a file relative to directory file descriptor

SYNOPSIS

```
#include <unistd.h>

int access(const char *path, int amode);
int faccessat(int fd, const char *path, int amode, int flag);
```

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 1003.1-200x, Section 4.4, File Access Permissions, except that where that description refers to execute permission for a process with appropriate privileges, an implementation may indicate success for X_OK even if execute permission is not granted to any user.

The *faccessat*( ) function shall be equivalent to the *access*( ) function, except in the case where *path* specifies a relative path. In this case the file whose accessibility is to be determined shall be located relative to the directory associated with the file descriptor *fd* instead of the current working directory. It is unspecified whether directory searches are permitted based on whether the file was opened with search permission or on the current permissions of the directory underlying the file descriptor.

If *faccessat*( ) is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior shall be identical to a call to *access*( ).

Values for *flag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in **<fcntl.h>**:

AT_EACCESS The checks for accessibility are performed using the effective user and group IDs instead of the real user and group ID as required in a call to *access*( ).

RETURN VALUE

Upon successful completion, these functions shall return 0. Otherwise, these functions shall return –1 and set *errno* to indicate the error.

ERRORS

These functions 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 *faccessat*( ) function shall fail if:

[EBADF] The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor.

These functions 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.

The *faccessat*( ) function may fail if:

[EINVAL] The value of the *flag* argument is not valid.

[ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

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.

The use of the AT_EACCESS value for *flag* enables functionality not available in *access*( ).

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 1003.1-200x 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 1003.1-200x.

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.

The purpose of the *faccessat*( ) function is to enable the checking of the accessibility of files in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to *access*( ), resulting in unspecified behavior. By opening a file descriptor for the target directory and using the *faccessat*( ) function it can be guaranteed that the file tested for accessibility is located relative to the desired directory.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod( ), *fstatat*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fcntl.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 [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.

Issue 7

Austin Group Interpretation 1003.1-2001 #046 is applied.

The *faccessat*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 2.

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 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

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 *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 arc cosine of *x*, in the range [0,π] 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_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

None.

FUTURE DIRECTIONS

None.

SEE ALSO

cos( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, 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.

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

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

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 1003.1-200x 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 *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 cosine of their argument.

MX 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.

MX 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:

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

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

cosh( ), *feclearexcept*( ), *fetestexcept*( ), the Base Definitions volume of IEEE Std 1003.1-200x, 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.

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

4180 **NAME**

4181 acosl — arc cosine functions

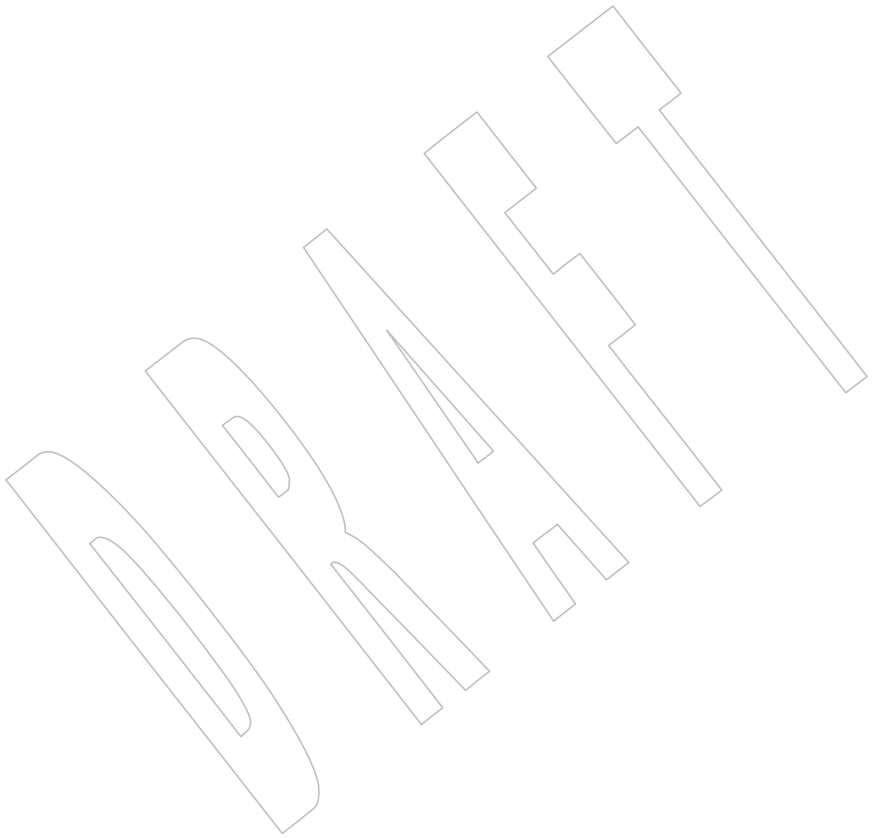
4182 **SYNOPSIS**

4183
```
#include <math.h>
```

4184
```
long double acosl(long double x);
```

4185 **DESCRIPTION**

4186 Refer to *acos( )*.



NAME

aio_cancel — cancel an asynchronous I/O request

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 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.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

aio_read( ), *aio_write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/10 is applied, removing the words "to the calling process" in the RETURN VALUE section. The term was unnecessary and precluded threads.

Issue 7

The *aio_cancel*( ) function is moved from the Asynchronous Input and Output option to the Base.

NAME

aio_error — retrieve errors status for an asynchronous I/O operation

SYNOPSIS

```
#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].

If the **aiocb** structure pointed to by *aiocbp* is not associated with an operation that has been scheduled, the results are undefined.

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 SIO *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

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

aio_cancel( ), *aio_fsync*( ), *aio_read*( ), *aio_return*( ), *aio_write*( ), *close*( ), *exec*, *exit*( ), *fork*( ), *lio_listio*( ), *lseek*( ), *read*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

Issue 7

Austin Group Interpretation 1003.1-2001 #045 is applied.

The *aio_error*( ) function is moved from the Asynchronous Input and Output option to the Base.

4280 **NAME**

4281 aio_fsync — asynchronous file synchronization

4282 **SYNOPSIS**

4283 ```
#include <aio.h>
```

4284 ```
int aio_fsync(int op, struct aiocb *aiocbp);
```

4285 **DESCRIPTION**

4286 The *aio_fsync*( ) function shall asynchronously force all I/O operations associated with the file
4287 indicated by the file descriptor *aio_fildes* member of the **aiocb** structure referenced by the *aiocbp*
4288 argument and queued at the time of the call to *aio_fsync*( ) to the synchronized I/O completion
4289 state. The function call shall return when the synchronization request has been initiated or
4290 queued to the file or device (even when the data cannot be synchronized immediately).

4291 If *op* is O_DSYNC, all currently queued I/O operations shall be completed as if by a call to
4292 *fdatasync*( ); that is, as defined for synchronized I/O data integrity completion. If *op* is O_SYNC,
4293 all currently queued I/O operations shall be completed as if by a call to *fsync*( ); that is, as
4294 defined for synchronized I/O file integrity completion. If the *aio_fsync*( ) function fails, or if the
4295 operation queued by *aio_fsync*( ) fails, then, as for *fsync*( ) and *fdatasync*( ), outstanding I/O
4296 operations are not guaranteed to have been completed.

4297 If *aio_fsync*( ) succeeds, then it is only the I/O that was queued at the time of the call to
4298 *aio_fsync*( ) that is guaranteed to be forced to the relevant completion state. The completion of
4299 subsequent I/O on the file descriptor is not guaranteed to be completed in a synchronized
4300 fashion.

4301 The *aiocbp* argument refers to an asynchronous I/O control block. The *aiocbp* value may be used
4302 as an argument to *aio_error*( ) and *aio_return*( ) in order to determine the error status and return
4303 status, respectively, of the asynchronous operation while it is proceeding. When the request is
4304 queued, the error status for the operation is [EINPROGRESS]. When all data has been
4305 successfully transferred, the error status shall be reset to reflect the success or failure of the
4306 operation. If the operation does not complete successfully, the error status for the operation shall
4307 be set to indicate the error. The *aio_sigevent* member determines the asynchronous notification to
4308 occur as specified in Section 2.4.1 when all operations have achieved synchronized I/O
4309 completion. All other members of the structure referenced by *aiocbp* are ignored. If the control
4310 block referenced by *aiocbp* becomes an illegal address prior to asynchronous I/O completion,
4311 then the behavior is undefined.

4312 If the *aio_fsync*( ) function fails or *aiocbp* indicates an error condition, data is not guaranteed to
4313 have been successfully transferred.

4314 **RETURN VALUE**

4315 The *aio_fsync*( ) function shall return the value 0 if the I/O operation is successfully queued;
4316 otherwise, the function shall return the value –1 and set *errno* to indicate the error.

4317 **ERRORS**

4318 The *aio_fsync*( ) function shall fail if:

4319 [EAGAIN] The requested asynchronous operation was not queued due to temporary
4320 resource limitations.

4321 [EBADF] The *aio_fildes* member of the **aiocb** structure referenced by the *aiocbp* argument
4322 is not a valid file descriptor open for writing.

4323 [EINVAL] This implementation does not support synchronized I/O for this file.

[EINVAL] 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*( ).

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fcntl( ), *fdatasync*( ), *fsync*( ), *open*( ), *read*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/11 is applied, removing the words "to the calling process" in the RETURN VALUE section. The term was unnecessary and precluded threads.

Issue 7

The *aio_fsync*( ) function is moved from the Asynchronous Input and Output option to the Base.

NAME

aio_read — asynchronous read from a file

SYNOPSIS

```
#include <aio.h>

int aio_read(struct aiocb *aiocbp);
```

DESCRIPTION

The *aio_read*( ) function shall read *aiocbp−>aio_nbytes* from the file associated with *aiocbp−>aio_fildes* into the buffer pointed to by *aiocbp−>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).

PIO If prioritized I/O is supported for this file, then the asynchronous operation shall be submitted at a priority equal to a base scheduling priority minus *aiocbp−>aio_reqprio*. If Thread Execution Scheduling is not supported, then the base scheduling priority is that of the calling process;
PIO TPS otherwise, the base scheduling priority is that of the calling thread.

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. 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 *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. After a successful call to enqueue an asynchronous I/O operation, the value of the file offset for the file is unspecified.

The *aio_sigevent* member specifies the notification which occurs when the request is completed.

The *aiocbp−>aio_lio_opcode* field shall be ignored by *aio_read*( ).

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.

Simultaneous asynchronous operations using the same *aiocbp* produce undefined results.

SIO 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_read*( ) function shall return the value zero 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,

PIO *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:

[EOVERFLOW] The file is a regular file, *aiobcp–>aio_nbytes* is greater than 0, and the starting offset in *aiobcp–>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

None.

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 1003.1-200x, **<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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/12 is applied, rewording the DESCRIPTION when prioritized I/O is supported to account for threads, and removing the words "to the calling process" in the RETURN VALUE section.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/13 is applied, updating the [EINVAL] error, so that detection of an [EINVAL] error for an invalid value of *aiocbp->aio_reqprio* is only required if the Prioritized Input and Output option is supported.

Issue 7

Austin Group Interpretation 1003.1-2001 #082 is applied.

The *aio_read*( ) function is moved from the Asynchronous Input and Output option to the Base.

NAME

aio_return — retrieve return status of an asynchronous I/O operation

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

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

aio_cancel( ), *aio_error*( ), *aio_fsync*( ), *aio_read*( ), *aio_write*( ), *close*( ), *exec*, *exit*( ), *fork*( ), *lio_listio*( ), *lseek*( ), *read*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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 made optional. This is for consistency with the DESCRIPTION.

Issue 7

The *aio_return*( ) function is moved from the Asynchronous Input and Output option to the Base.

NAME

aio_suspend — wait for an asynchronous I/O request

SYNOPSIS

```
#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.

MON 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

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

aio_read( ), *aio_write*( ), *lio_listio*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

Issue 7

The *aio_suspend*( ) function is moved from the Asynchronous Input and Output option to the Base.

NAME

aio_write — asynchronous write to a file

SYNOPSIS

```
#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.

PIO If prioritized I/O is supported for this file, then the asynchronous operation shall be submitted at a priority equal to a base scheduling priority minus *aiocbp−>aio_reqprio*. If Thread Execution Scheduling is not supported, then the base scheduling priority is that of the calling process;
PIO TPS otherwise, the base scheduling priority is that of the calling thread.

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 *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 *aio_sigevent* member specifies the notification which occurs when the request is completed.

The *aiocbp−>aio_lio_opcode* field shall be ignored by *aio_write*( ).

Simultaneous asynchronous operations using the same *aiocbp* produce undefined results.

SIO 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 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,

PIO *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, *aiobcp–>aio_nbytes* is greater than 0, and the starting offset in *aiobcp–>aio_offset* is at or beyond the offset maximum in the open file description associated with *aiocbp–>aio_fildes*.

EXAMPLES

None.

APPLICATION USAGE

None.

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 1003.1-200x, **<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 *aiocbp–>aio_fildes*.

- The [EFBIG] error is added as part of the large file support extensions.

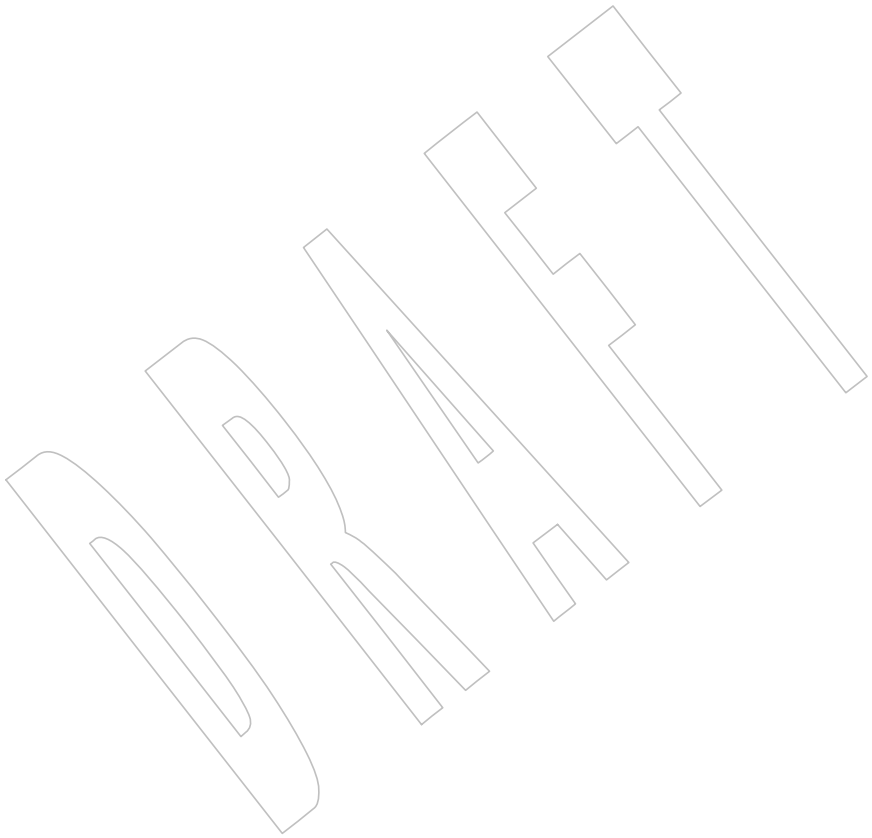
IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/14 is applied, rewording the DESCRIPTION when prioritized I/O is supported to account for threads, and removing the words "to the calling process" in the RETURN VALUE section.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/15 is applied, updating the [EINVAL] error, so that detection of an [EINVAL] error for an invalid value of *aiocbp->aio_reqprio* is only required if the Prioritized Input and Output option is supported.

Issue 7

Austin Group Interpretation 1003.1-2001 #082 is applied.

The *aio_write*( ) function is moved from the Asynchronous Input and Output option to the Base.



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 *setitimer*( ) 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 image of the old process.

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 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**.

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-specific maximum value. This maximum is large enough (to the order of several months) that the effect is not noticeable.

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*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/16 is applied, replacing "an implementation-defined maximum value" with "an implementation-specific maximum value" in the RATIONALE.

NAME

alphasort, scandir — scan a directory

SYNOPSIS

```
#include <dirent.h>

int alphasort(const struct dirent **d1, const struct dirent **d2);
int scandir(const char *dir, struct dirent ***namelist,
    int (*sel)(const struct dirent *),
    int (*compar)(const struct dirent **, const struct dirent **));
```

DESCRIPTION

The *alphasort*( ) function can be used as the comparison function for the *scandir*( ) function to sort the directory entries into alphabetical order. Sorting happens as if by calling the *strcoll*( ) function on the *d_name* element of the **dirent** structures passed as the two parameters. Its parameters are the two directory entries, *d1* and *d2*, to compare.

The *scandir*( ) function shall scan the directory *dir*, calling the function referenced by *sel* on each directory entry. Entries for which the function referenced by *sel* returns non-zero shall be stored in strings allocated as if by a call to *malloc*( ), and sorted using *qsort*( ) with the comparison function *compar*, and collected in array *namelist* which shall be allocated as if by a call to *malloc*( ). If *sel* is a null pointer, all entries shall be selected.

RETURN VALUE

Upon successful completion, the *alphasort*( ) function shall return an integer greater than, equal to, or less than 0, according to whether the name of the directory entry pointed to by *d1* is lexically greater than, equal to, or less than the directory pointed to by *d2* when both are interpreted as appropriate to the current locale. There is no return value reserved to indicate an error.

Upon successful completion, the *scandir*( ) function shall return the number of entries in the array and a pointer to the array through the parameter *namelist*. Otherwise, the *scandir*( ) function shall return –1.

ERRORS

The *scandir*( ) function shall fail if:

[EACCES] Search permission is denied for the component of the path prefix of *dir* or read permission is denied for *dir*.

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

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

[ENOENT] A component of *dir* does not name an existing directory or *dir* is an empty string.

[ENOMEM] Insufficient storage space is available.

[ENOTDIR] A component of *dir* is not a directory.

The *scandir*( ) function may fail if:

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *dir* 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 *dir* argument, the length of the substituted pathname string exceeded {PATH_MAX}.

[ENFILE] Too many files are currently open in the system.

EXAMPLES

An example to print the files in the current directory:

```
#include <dirent.h>
#include <stdio.h>
...
struct dirent **namelist;
int i,n;

    n = scandir(".", &namelist, 0, alphasort);
    if (n < 0)
        perror("scandir");
    else {
        for (i = 0; i < n; i++) {
            printf("%s\n", namelist[i]->d_name);
            free(namelist[i]);
            }
        }
    free(namelist);
...
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

malloc( ), *qsort( )*, *strcoll( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<dirent.h>**

CHANGE HISTORY

First released in Issue 7.

NAME

asctime, asctime_r — convert date and time to a string

SYNOPSIS

OB
```
#include <time.h>

char *asctime(const struct tm *timeptr);
```
OB CX
```
char *asctime_r(const struct tm *restrict tm, char *restrict buf);
```

DESCRIPTION

CX For *asctime*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x 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\n\0
```

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] = {
        "Jan", "Feb", "Mar", "Apr", "May", "Jun",
        "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;
}
```

However, the behavior is undefined if *timeptr–>tm_wday* or *timeptr–>tm_mon* are not within the normal ranges as defined in **<time.h>**, or if *timeptr–>tm_year* exceeds {INT_MAX}–1990, or if the above algorithm would attempt to generate more than 26 bytes of output (including the terminating null).

The **tm** structure is defined in the **<time.h>** header.

CX The *asctime*( ), *ctime*( ), *gmtime*( ), and *localtime*( ) functions shall return values in one of two static objects: a broken-down time structure and an array of type **char**. Execution of any of the functions may overwrite the information returned in either of these objects by any of the other functions.

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

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*( ), and with the same undefined behavior when input or output is out of range) 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

CX Upon successful completion, *asctime*( ) shall return a pointer to the string. If the function is unsuccessful, it shall return NULL.

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

These functions are included only for compatibility with older implementations. They have undefined behavior if the resulting string would be too long, so the use of these functions should be discouraged. On implementations that do not detect output string length overflow, it is possible to overflow the output buffers in such a way as to cause applications to fail, or possible system security violations. Also, these functions do not support localized date and time formats. To avoid these problems, applications should use *strftime*( ) to generate strings from broken-down times.

Values for the broken-down time structure can be obtained by calling *gmtime*( ) or *localtime*( ).

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

The standards developers decided to mark the *asctime*( ) and *asctime_r*( ) functions obsolescent even though they are in the ISO C standard due to the possibility of buffer overflow. The ISO C standard also provides the *strftime*( ) function which can be used to avoid these problems.

FUTURE DIRECTIONS

These functions may be removed in a future version.

SEE ALSO

clock( ), *ctime*( ), *difftime*( ), *gmtime*( ), *localtime*( ), *mktime*( ), *strftime*( ), *strptime*( ), *time*( ), *utime*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

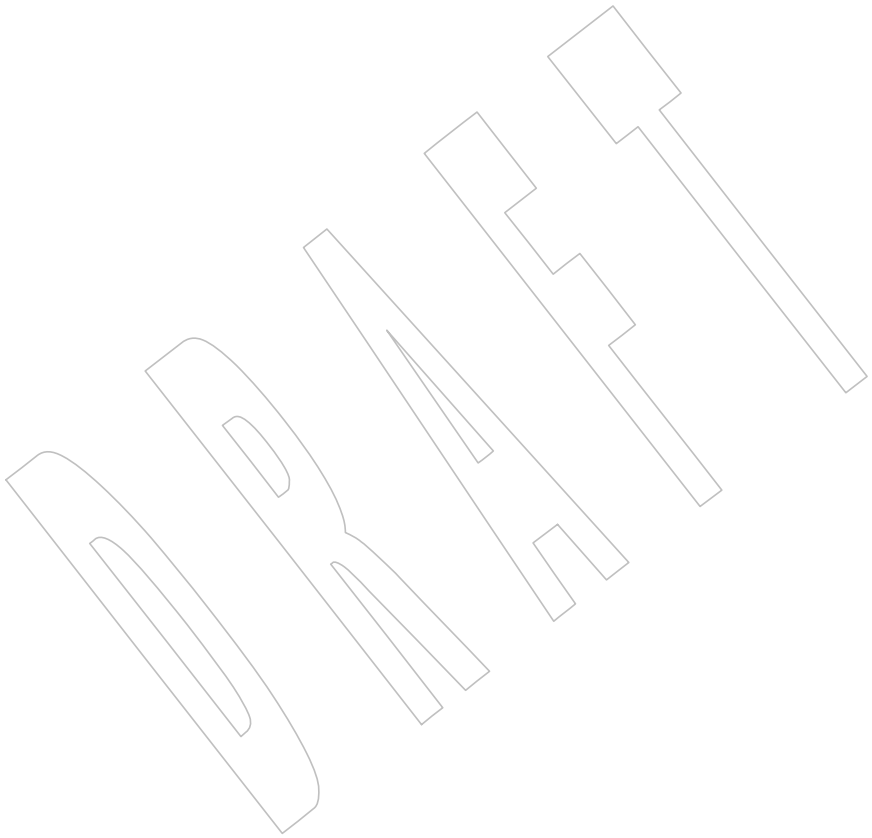
The **restrict** keyword is added to the *asctime_r*( ) prototype for alignment with the ISO/IEC 9899: 1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/17 is applied, adding the CX extension in the RETURN VALUE section requiring that if the *asctime*( ) function is unsuccessful it returns NULL.

Issue 7

Austin Group Interpretation 1003.1-2001 #053 is applied, marking these functions obsolescent.

The *asctime_r*( ) function is moved from the Thread-Safe Functions option to the Base.



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

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 1003.1-200x defers to the ISO C standard.

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 *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 arc sine of *x*, in the range $[-\pi/2,\pi/2]$ 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 ±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:

MX 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 *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 (*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

feclearexcept( ), *fetestexcept( )*, *isnan( )*, *sin( )*, the Base Definitions volume of IEEE Std 1003.1-200x, 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.

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

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

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 1003.1-200x 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 *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 sine of their argument.

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

If *x* is ±0, or ±Inf, *x* shall be returned.

If *x* is subnormal, a range error may occur and *x* should be returned.

ERRORS

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 (*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

feclearexcept( ), *fetestexcept*( ), *sinh*( ), the Base Definitions volume of IEEE Std 1003.1-200x, 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 *asinh*( ) function is no longer marked as an extension.

The *asinhf*( ) and *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.

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

5004 **NAME**

5005 asinl — arc sine function

5006 **SYNOPSIS**

5007
```
#include <math.h>
```

5008
```
long double asinl(long double x);
```

5009 **DESCRIPTION**

5010 Refer to *asin( )*.

NAME

assert — insert program diagnostics

SYNOPSIS

```
#include <assert.h>

void assert(scalar expression);
```

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 1003.1-200x 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 1003.1-200x, **<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

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 1003.1-200x defers to the ISO C standard.

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 *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 arc tangent of *x* in the range $[-\pi/2,\pi/2]$ radians.

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

If *x* is ±0, *x* shall be returned.

If *x* is ±Inf, $\pm\pi/2$ shall be returned.

If *x* is subnormal, a range error may occur and *x* should be returned.

ERRORS

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 (*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 1003.1-200x, 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 *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.

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

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

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 1003.1-200x defers to the ISO C standard.

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 *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 arc tangent of y/x in the range $[-\pi,\pi]$ radians.

If y is ±0 and x is < 0, $\pm\pi$ shall be returned.

If y is ±0 and x is > 0, ±0 shall be returned.

If y is < 0 and x is ±0, $-\pi/2$ shall be returned.

If y is > 0 and x is ±0, $\pi/2$ shall be returned.

If x is 0, a pole error shall not occur.

MX If either x or y is NaN, a NaN shall be returned.

If the result underflows, a range error may occur and y/x should be returned.

If y is ±0 and x is –0, $\pm\pi$ shall be returned.

If y is ±0 and x is +0, ±0 shall be returned.

For finite values of $\pm y > 0$, if x is –Inf, $\pm\pi$ shall be returned.

For finite values of $\pm y > 0$, if x is +Inf, ±0 shall be returned.

For finite values of x, if y is ±Inf, $\pm\pi/2$ shall be returned.

If y is ±Inf and x is –Inf, $\pm 3\pi/4$ shall be returned.

If y is ±Inf and x is +Inf, $\pm\pi/4$ shall be returned.

If both arguments are 0, a domain error shall not occur.

ERRORS

These functions may fail if:

MX 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

Converting Cartesian to Polar Coordinates System

The function below uses *atan2*( ) to convert a 2d vector expressed in cartesian coordinates (*x*,*y*) to the polar coordinates (*rho*,*theta*). There are other ways to compute the angle *theta*, using *asin*( ) *acos*( ), or *atan*( ). However, *atan2*( ) presents here two advantages:

- The angle's quadrant is automatically determined.
- The singular cases (0,*y*) are taken into account.

Finally, this example uses *hypot*( ) rather than *sqrt*( ) since it is better for special cases; see *hypot*( ) for more information.

```
#include <math.h>

void
cartesian_to_polar(const double x, const double y,
                   double *rho, double *theta
    )
{
    *rho   = hypot (x,y); /* better than sqrt(x*x+y*y) */
    *theta = atan2 (y,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

acos( ), *asin*( ), *atan*( ), *feclearexcept*( ), *fetestexcept*( ), *hypot*( ), *isnan*( ), *sqrt*( ), *tan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, 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, and the IEC 60559: 1989 standard floating-point extensions over the ISO/IEC 9899: 1999 standard are marked.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/18 is applied, adding to the EXAMPLES section.

5189 **NAME**

5190 atanf — arc tangent function

5191 **SYNOPSIS**

5192
```
#include <math.h>
```

5193
```
float atanf(float x);
```

5194 **DESCRIPTION**

5195 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

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 1003.1-200x 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.

MX For finite $|x|>1$, 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 ±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:

MX Domain Error — The *x* argument is finite and not in the range [–1,1], or 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 *x* argument is ±1.

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.

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 (*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

feclearexcept( ), *fetestexcept( )*, *tanh( )*, the Base Definitions volume of IEEE Std 1003.1-200x, 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 *atanh( )* function is no longer marked as an extension.

The *atanhf( )* and *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.

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

5267 **NAME**

5268 atanl — arc tangent function

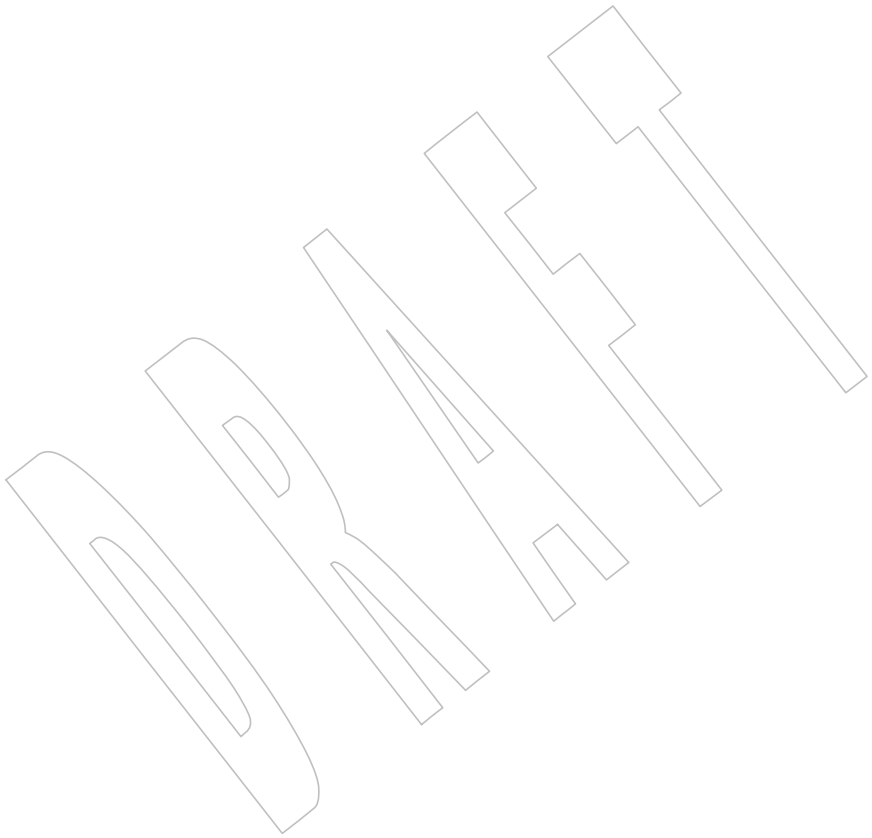
5269 **SYNOPSIS**

5270
```
#include <math.h>
```

5271
```
long double atanl(long double x);
```

5272 **DESCRIPTION**

5273 Refer to *atan( )*.



NAME

atexit — register a function to run at process termination

SYNOPSIS

```
#include <stdlib.h>

int atexit(void (*func)(void));
```

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 1003.1-200x defers to the ISO C standard.

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.

EXAMPLES

None.

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*( ).

Since the behavior is undefined if the *exit*( ) function is called more than once, portable applications calling *atexit*( ) must ensure that the *exit*( ) function is not called at normal process termination when all functions registered by the *atexit*( ) function are called.

All functions registered by the *atexit*( ) function are called at normal process termination, which occurs by a call to the *exit*( ) function or a return from *main*( ) or on the last thread termination, when the behavior is as if the implementation called *exit*( ) with a zero argument at thread termination time.

If, at normal process termination, a function registered by the *atexit*( ) function is called and a portable application needs to stop further *exit*( ) processing, it must call the *_exit*( ) function or the *_Exit*( ) function or one of the functions which cause abnormal process termination.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *exit*( ), *sysconf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/19 is applied, adding further clarification to the APPLICATION USAGE section.

NAME

atof — convert a string to a double-precision number

SYNOPSIS

```
#include <stdlib.h>

double atof(const char *str);
```

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 1003.1-200x 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.

EXAMPLES

None.

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 1003.1-200x, **<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

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 1003.1-200x 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

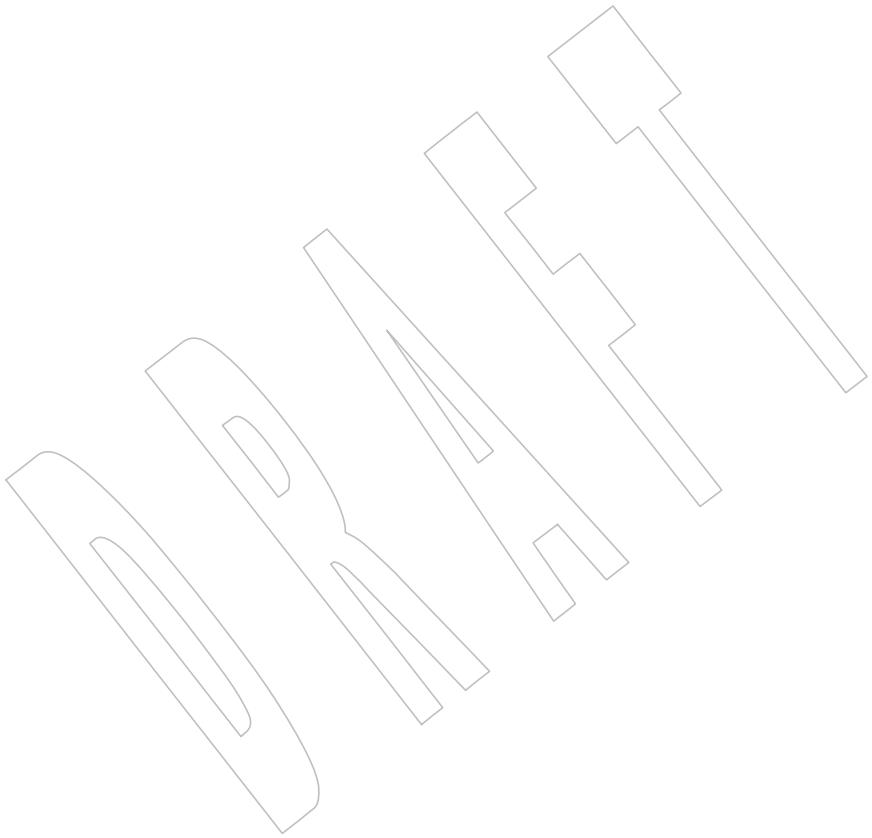
None.

SEE ALSO

strtol( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

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



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

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 1003.1-200x defers to the ISO C standard.

The call *atol*(*str*) shall be equivalent to:

```
strtol(str, (char **)NULL, 10)
```

The call *atoll*(*nptr*) 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 1003.1-200x, **<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.

Issue 7

SD5-XSH-ERN-61 is applied, correcting the DESCRIPTION of *atoll*( ).

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 pointed to by *path* consists entirely of the `'/'` character, *basename*( ) shall return a pointer to the string `"/"`. If the string pointed to by *path* 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 thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

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.

Input String	Output String
`"/usr/lib"`	`"lib"`
`"/usr/"`	`"usr"`
`"/"`	`"/"`
`"///"`	`"/"`
`"//usr//lib//"`	`"lib"`

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

dirname( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<libgen.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x, *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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/20 is applied, changing the DESCRIPTION to make it clear that the string referenced is the string pointed to by *path.*

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.

[ENOBUFS] Insufficient resources were available to complete the call.

[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.

[EDESTADDRREQ] or [EISDIR]
The *address* argument is a null pointer.

[EIO] An I/O error occurred.

[ELOOP] A loop exists in symbolic links encountered during resolution of the pathname in *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 *address* is not a directory.

[EROFS] The name would reside on a read-only file system.

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}.

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 1003.1-200x, **<sys/socket.h>**

CHANGE HISTORY

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

Issue 7

Austin Group Interpretation 1003.1-2001 #044 is applied, changing the "may fail" [ENOBUFS] error to become a "shall fail" error.

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

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 1003.1-200x defers to the ISO C standard.

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*. If the *nel* argument has the value zero, the comparison function pointed to by *compar* shall not be called and no match shall be found.

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 comparison function pointed to by *compar* does not alter the contents of the array. The implementation may reorder elements of the array between calls to the comparison function, but shall not alter the contents of any individual element.

The implementation shall ensure that the first argument is always a pointer to the key.

When the same objects (consisting of width bytes, irrespective of their current positions in the array) are passed more than once to the comparison function, the results shall be consistent with one another. That is, the same object shall always compare the same way with the key.

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.

ERRORS

No errors are defined.

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 {                   /* These are stored in the table. */
```



```
        char *string;
        int length;
    };
    struct node table[TABSIZE];    /* Table to be searched. */
        .
        .
        .
    {
        struct node *node_ptr, node;
        /* Routine to compare 2 nodes. */
        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

The requirement that the second argument (hereafter referred to as *p*) to the comparison function is a pointer to an element of the array implies that for every call all of the following expressions are non-zero:

```
((char *)p – (char *(base) % width == 0
(char *)p >= (char *)base
(char *)p < (char *)base + nel * width
```

FUTURE DIRECTIONS

None.

SEE ALSO

hcreate( ), *lsearch*( ), *qsort*( ), *tsearch*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

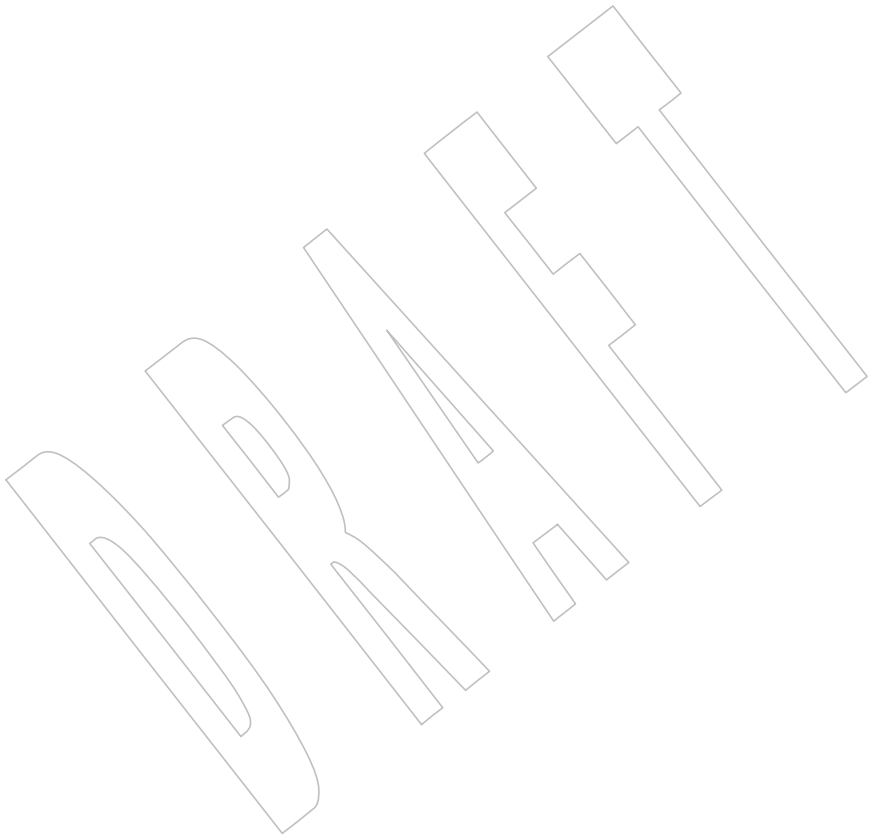
CHANGE HISTORY

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

Issue 6

The normative text is updated to avoid use of the term "must" for application requirements.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/11 is applied, adding to the DESCRIPTION the last sentence of the first non-shaded paragraph, and the following three paragraphs. The RATIONALE section is also updated. These changes are for alignment with the ISO C standard.



NAME

btowc — single byte to wide character conversion

SYNOPSIS

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

wint_t btowc(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 1003.1-200x 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 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

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

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 1003.1-200x 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 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

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

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 1003.1-200x 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, π] along the real axis.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ccos( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

NAME

cacosh, cacoshf, cacoshl — complex arc hyperbolic cosine functions

SYNOPSIS

```
#include <complex.h>

double complex cacosh(double complex z);
float complex cacoshf(float complex z);
long double complex cacoshl(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 1003.1-200x defers to the ISO C standard.

These functions shall compute the complex arc hyperbolic cosine of *z*, with a branch cut at values less than 1 along the real axis.

RETURN VALUE

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 $[-i\pi, +i\pi]$ along the imaginary axis.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ccosh( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

5806 **NAME**

5807 cacosl — complex arc cosine functions

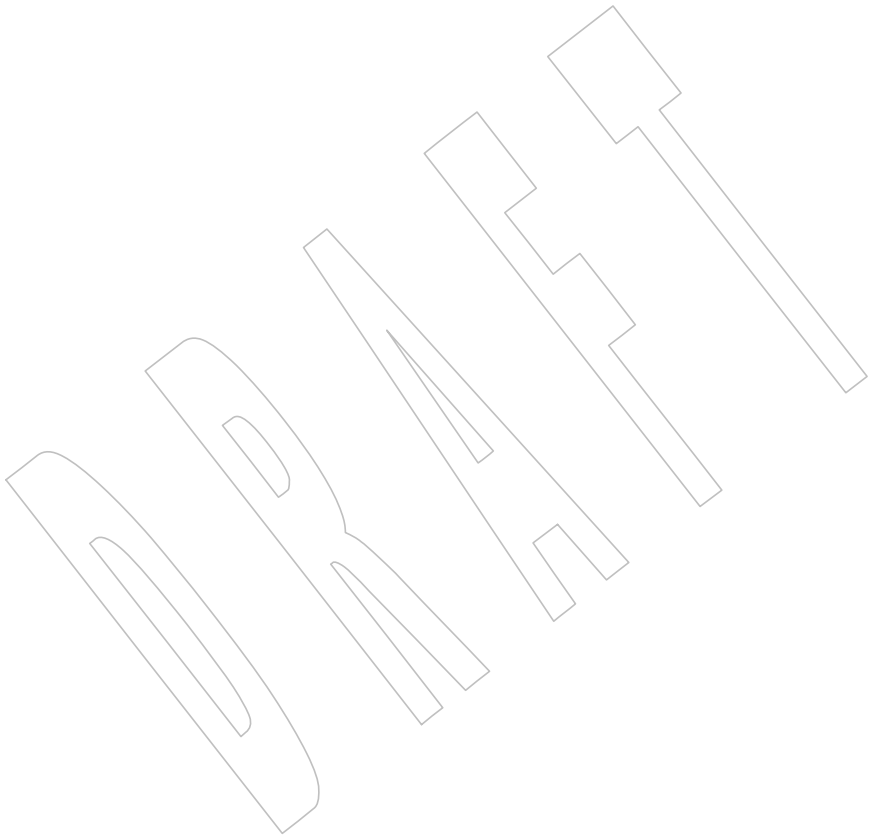
5808 **SYNOPSIS**

5809
```
#include <complex.h>
```

5810
```
long double complex cacosl(long double complex z);
```

5811 **DESCRIPTION**

5812 Refer to *cacos( )*.



NAME

calloc — a memory allocator

SYNOPSIS

```
#include <stdlib.h>

void *calloc(size_t nelem, size_t elsize);
```

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 1003.1-200x 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 CX and set *errno* to indicate the error.

ERRORS

The *calloc*( ) function shall fail if:

CX [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 1003.1-200x, **<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

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 1003.1-200x defers to the ISO C standard.

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 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

5887 **NAME**

5888 casin, casinf, casinl — complex arc sine functions

5889 **SYNOPSIS**

5890
```
#include <complex.h>
```

5891
```
double complex casin(double complex z);
```
5892
```
float complex casinf(float complex z);
```
5893
```
long double complex casinl(long double complex z);
```

5894 **DESCRIPTION**

5895 CX The functionality described on this reference page is aligned with the ISO C standard. Any
5896 conflict between the requirements described here and the ISO C standard is unintentional. This
5897 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

5898 These functions shall compute the complex arc sine of z, with branch cuts outside the interval
5899 [−1, +1] along the real axis.

5900 **RETURN VALUE**

5901 These functions shall return the complex arc sine value, in the range of a strip mathematically
5902 unbounded along the imaginary axis and in the interval $[-\pi/2, +\pi/2]$ along the real axis.

5903 **ERRORS**

5904 No errors are defined.

5905 **EXAMPLES**

5906 None.

5907 **APPLICATION USAGE**

5908 None.

5909 **RATIONALE**

5910 None.

5911 **FUTURE DIRECTIONS**

5912 None.

5913 **SEE ALSO**

5914 *csin*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

5915 **CHANGE HISTORY**

5916 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

NAME

casinh, casinhf, casinhl — complex arc hyperbolic sine functions

SYNOPSIS

```
#include <complex.h>

double complex casinh(double complex z);
float complex casinhf(float 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 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

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 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

NAME

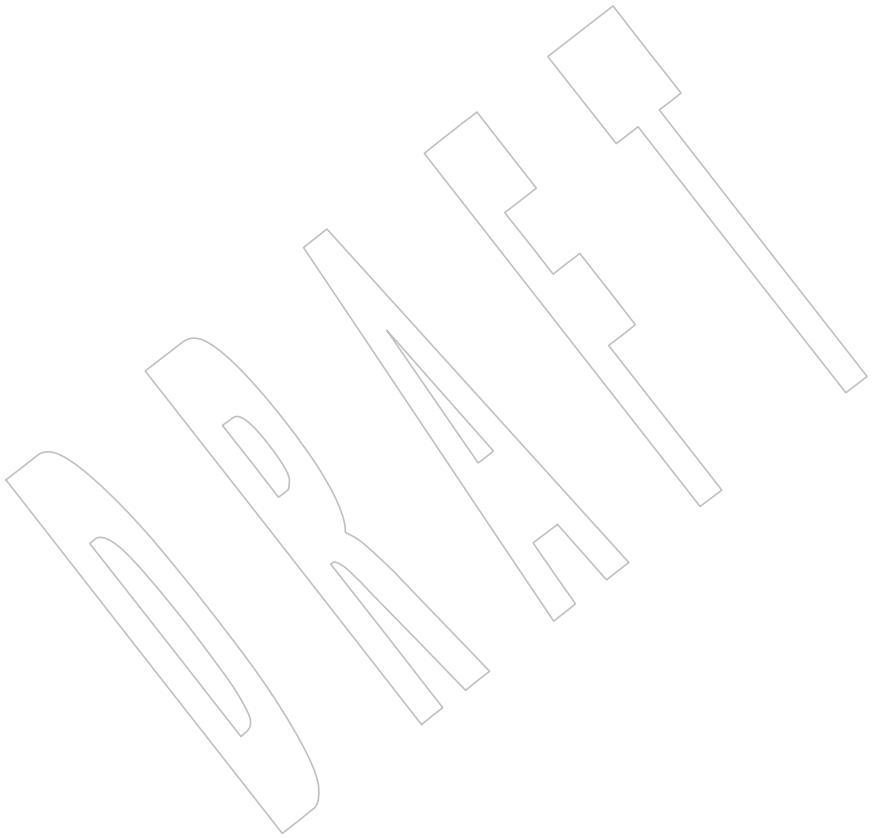
casinl — complex arc sine functions

SYNOPSIS

```
#include <complex.h>

long double complex casinl(long double complex z);
```

DESCRIPTION

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 1003.1-200x 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 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

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

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 1003.1-200x 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

ctanh( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

6017 **NAME**

6018 catanl — complex arc tangent functions

6019 **SYNOPSIS**

6020
```
#include <complex.h>
```

6021
```
long double complex catanl(long double complex z);
```

6022 **DESCRIPTION**

6023 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 1003.1-200x, **<nl_types.h>**

CHANGE HISTORY

First released in Issue 2.

Issue 7

The *catclose*( ) function is moved from the XSI option to the Base.

6053 **NAME**

6054 catgets — read a program message

6055 **SYNOPSIS**

6056
```
#include <nl_types.h>
```

6057
```
char *catgets(nl_catd catd, int set_id, int msg_id, const char *s);
```

6058 **DESCRIPTION**

6059 The *catgets*( ) function shall attempt to read message *msg_id*, in set *set_id*, from the message
6060 catalog identified by *catd*. The *catd* argument is a message catalog descriptor returned from an
6061 earlier call to *catopen*( ). The results are undefined if *catd* is not a value returned by *catopen*( ) for
6062 a message catalog still open in the process. The *s* argument points to a default message string
6063 which shall be returned by *catgets*( ) if it cannot retrieve the identified message.

6064 The *catgets*( ) function need not be thread-safe. A function that is not required to be thread-safe is
6065 not required to be reentrant.

6066 **RETURN VALUE**

6067 If the identified message is retrieved successfully, *catgets*( ) shall return a pointer to an internal
6068 buffer area containing the null-terminated message string. If the call is unsuccessful for any
6069 reason, *s* shall be returned and *errno* shall be set to indicate the error.

6070 **ERRORS**

6071 The *catgets*( ) function shall fail if:

6072 [EINTR] The read operation was terminated due to the receipt of a signal, and no data
6073 was transferred.

6074 [ENOMSG] The message identified by *set_id* and *msg_id* is not in the message catalog.

6075 The *catgets*( ) function may fail if:

6076 [EBADF] The *catd* argument is not a valid message catalog descriptor open for reading.

6077 [EBADMSG] The message identified by *set_id* and *msg_id* in the specified message catalog
6078 did not satisfy implementation-defined security criteria.

6079 [EINVAL] The message catalog identified by *catd* is corrupted.

6080 **EXAMPLES**

6081 None.

6082 **APPLICATION USAGE**

6083 None.

6084 **RATIONALE**

6085 None.

6086 **FUTURE DIRECTIONS**

6087 None.

6088 **SEE ALSO**

6089 *catclose*( ), *catopen*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<nl_types.h>**

6090 **CHANGE HISTORY**

6091 First released in Issue 2.

Issue 5

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.

Issue 7

Austin Group Interpretation 1003.1-2001 #044 is applied, changing the "may fail" [EINTR] and [ENOMSG] errors to become "shall fail" errors, updating the RETURN VALUE section, and updating the DESCRIPTION to note that: "The results are undefined if *catd* is not a value returned by *catopen*( ) for a message catalog still open in the process.

The *catgets*( ) function is moved from the XSI option to the Base.

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 1003.1-200x, 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 1003.1-200x, 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 (**nl_catd**) –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] All file descriptors available to the process are currently open.

[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.

[ENOMEM] Insufficient storage space is available.

[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 1003.1-200x, **<fcntl.h>**, **<nl_types.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x

CHANGE HISTORY

First released in Issue 2.

Issue 7

SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

The *catopen*( ) function is moved from the XSI option to the Base.

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

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 1003.1-200x 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*.

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

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 1003.1-200x, **<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.

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

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

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 1003.1-200x defers to the ISO C standard.

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 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

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

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 1003.1-200x 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 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

6267 **NAME**

6268 ccosl — complex cosine functions

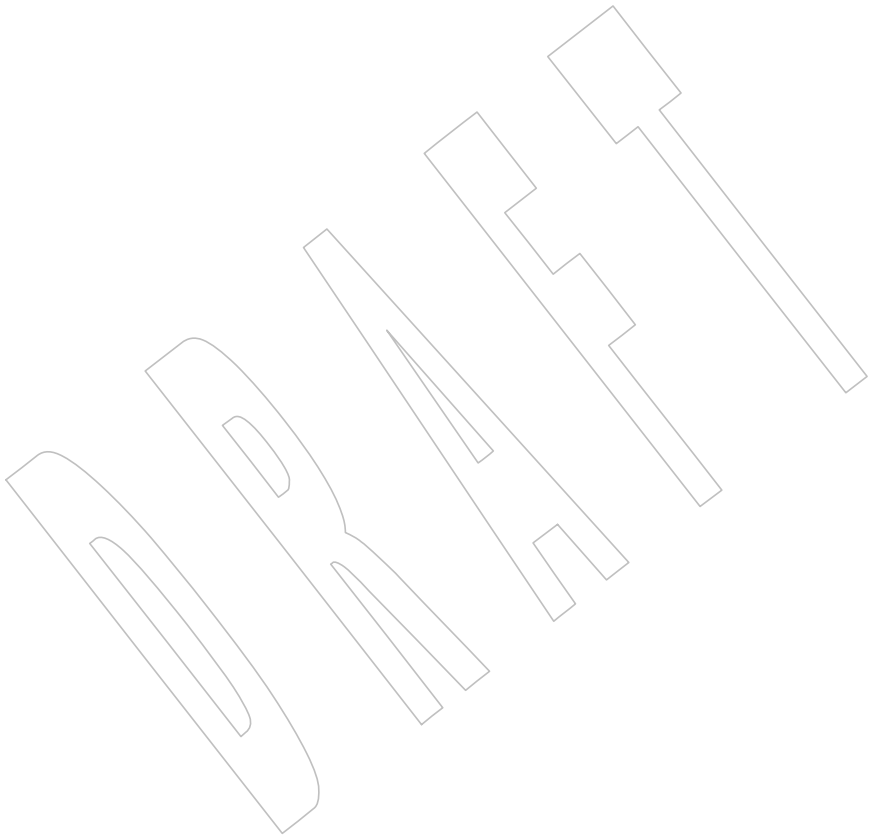
6269 **SYNOPSIS**

6270
```
#include <complex.h>
```

6271
```
long double complex ccosl(long double complex z);
```

6272 **DESCRIPTION**

6273 Refer to *ccos( )*.



6274 **NAME**

6275 ceil, ceilf, ceill — ceiling value function

6276 **SYNOPSIS**

6277
```
#include <math.h>

double ceil(double x);
float ceilf(float x);
long double ceill(long double x);
```

6281 **DESCRIPTION**

6282 CX The functionality described on this reference page is aligned with the ISO C standard. Any
6283 conflict between the requirements described here and the ISO C standard is unintentional. This
6284 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

6285 These functions shall compute the smallest integral value not less than *x*.

6286 An application wishing to check for error situations should set *errno* to zero and call
6287 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
6288 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
6289 zero, an error has occurred.

6290 **RETURN VALUE**

6291 Upon successful completion, *ceil*( ), *ceilf*( ), and *ceill*( ) shall return the smallest integral value not
6292 less than *x*, expressed as a type **double**, **float**, or **long double**, respectively.

6293 MX If *x* is NaN, a NaN shall be returned.

6294 If *x* is ±0 or ±Inf, *x* shall be returned.

6295 XSI If the correct value would cause overflow, a range error shall occur and *ceil*( ), *ceilf*( ), and *ceill*( )
6296 shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

6297 **ERRORS**

6298 These functions shall fail if:

6299 XSI Range Error The result overflows.

6300 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
6301 then *errno* shall be set to [ERANGE]. If the integer expression
6302 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
6303 floating-point exception shall be raised.

6304 **EXAMPLES**

6305 None.

6306 **APPLICATION USAGE**

6307 The integral value returned by these functions need not be expressible as an **int** or **long**. The
6308 return value should be tested before assigning it to an integer type to avoid the undefined
6309 results of an integer overflow.

6310 The *ceil*( ) function can only overflow when the floating-point representation has
6311 DBL_MANT_DIG > DBL_MAX_EXP.

6312 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
6313 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

6314 **RATIONALE**

6315 None.

6316 **FUTURE DIRECTIONS**

6317 None.

6318 **SEE ALSO**

6319 *feclearexcept( )*, *fetestexcept( )*, *floor( )*, *isnan( )*, the Base Definitions volume of IEEE Std 1003.1-200x,
6320 Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

6321 **CHANGE HISTORY**

6322 First released in Issue 1. Derived from Issue 1 of the SVID.

6323 **Issue 5**

6324 The DESCRIPTION is updated to indicate how an application should check for an error. This
6325 text was previously published in the APPLICATION USAGE section.

6326 **Issue 6**

6327 The *ceilf( )* and *ceill( )* functions are added for alignment with the ISO/IEC 9899: 1999 standard.

6328 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
6329 revised to align with the ISO/IEC 9899: 1999 standard.

6330 IEC 60559: 1989 standard floating-point extensions over the ISO/IEC 9899: 1999 standard are
6331 marked.

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

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 1003.1-200x defers to the ISO C standard.

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 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

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 as 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 1003.1-200x 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 1003.1-200x 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 1003.1-200x 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 1003.1-200x, 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 1003.1-200x, 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 1003.1-200x, 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 1003.1-200x, 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 1003.1-200x, **<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, fchmodat — change mode of a file relative to directory file descriptor

SYNOPSIS

```
#include <sys/stat.h>

int chmod(const char *path, mode_t mode);
int fchmodat(int fd, const char *path, mode_t mode, int flag);
```

DESCRIPTION

XSI The *chmod*( ) function shall change S_ISUID, S_ISGID, S_ISVTX, and the file permission bits of the file named by the pathname pointed to by the *path* argument to the corresponding bits in the *mode* argument. The application shall ensure that the effective user ID of the process matches the owner of the file or the process has appropriate privileges in order to do this.

XSI S_ISUID, S_ISGID,

[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 *fchmodat*( ) function shall fail if:

[EBADF] The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

These functions 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}.

The *fchmodat*( ) function may fail if:

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

[ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

[EOPNOTSUPP] The AT_SYMLINK_NOFOLLOW bit is set in the *flag* argument, *path* names a symbolic link, and the system does not support changing the mode of a symbolic link.

EXAMPLES

Setting Read Permissions for User, Group, and Others

The following example sets read permissions for the owner, group, and others.

```
#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.

```
#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.

```
#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.

```
#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 1003.1-200x 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.

The purpose of the *fchmodat*( ) function is to enable changing the mode of files in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to *chmod*( ), resulting in unspecified behavior. By opening a file descriptor for the target directory and using the *fchmodat*( ) function it can be guaranteed that the changed file is located relative to the desired directory. Some implementations might allow changing the mode of symbolic links. This is not supported by the interfaces in the POSIX specification. Systems with such support provide an interface named *lchmod*( ). To support such implementations *fchmodat*( ) has a *flag* parameter.

FUTURE DIRECTIONS

None.

SEE ALSO

access( ), *chown*( ), *exec*, *fstatat*( ), *mkdir*( ), *mkfifo*( ), *mknod*( ), *open*( ), *statvfs*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fcntl.h>**, **<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 normative text is updated to avoid use of the term "must" for application requirements.

Issue 7

The *fchmodat*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 2.

NAME

chown, fchownat — change owner and group of a file relative to directory file descriptor

SYNOPSIS

```
#include <unistd.h>

int chown(const char *path, uid_t owner, gid_t group);
int fchownat(int fd, const char *path, uid_t owner, gid_t group,
    int flag);
```

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.

The *fchownat*( ) function shall be equivalent to the *chown*( ) and *lchown*( ) functions except in the case where *path* specifies a relative path. In this case the file to be changed is determined relative to the directory associated with the file descriptor *fd* instead of the current working directory. It is unspecified whether directory searches are permitted based on whether the file was opened with search permission or on the current permissions of the directory underlying the file descriptor.

Values for *flag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in **<fcntl.h>**:

AT_SYMLINK_NOFOLLOW
: If *path* names a symbolic link, ownership of the symbolic link is changed.

If *fchownat*( ) is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior shall be identical to a call to *chown*( ) or *lchown*( ) respectively, depending on whether or not the AT_SYMLINK_NOFOLLOW bit is set in the *flag* argument.

6759 **RETURN VALUE**

6760 Upon successful completion, these functions shall return 0. Otherwise, these functions shall
6761 return –1 and set *errno* to indicate the error. If –1 is returned, no changes are made in the user ID
6762 and group ID of the file.

6763 **ERRORS**

6764 These functions shall fail if:

6765 [EACCES] Search permission is denied on a component of the path prefix.

6766 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
6767 argument.

6768 [ENAMETOOLONG]
6769 The length of the *path* argument exceeds {PATH_MAX} or a pathname
6770 component is longer than {NAME_MAX}.

6771 [ENOTDIR] A component of the path prefix is not a directory.

6772 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

6773 [EPERM] The effective user ID does not match the owner of the file, or the calling
6774 process does not have appropriate privileges and
6775 _POSIX_CHOWN_RESTRICTED indicates that such privilege is required.

6776 [EROFS] The named file resides on a read-only file system.

6777 The *fchownat*( ) function shall fail if:

6778 [EBADF] The *path* argument does not specify an absolute path and the *fd* argument is
6779 neither AT_FDCWD nor a valid file descriptor open for searching.

6780 These functions may fail if:

6781 [EIO] An I/O error occurred while reading or writing to the file system.

6782 [EINTR] The *chown*( ) function was interrupted by a signal which was caught.

6783 [EINVAL] The owner or group ID supplied is not a value supported by the
6784 implementation.

6785 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
6786 resolution of the *path* argument.

6787 [ENAMETOOLONG]
6788 As a result of encountering a symbolic link in resolution of the *path* argument,
6789 the length of the substituted pathname string exceeded {PATH_MAX}.

6790 The *fchownat*( ) function may fail if:

6791 [EINVAL] The value of the *flag* argument is not valid.

6792 [ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a
6793 file descriptor associated with a directory.

6794 [EOPNOTSUPP] The *path* argument names a symbolic link and the implementation does not
6795 support setting the owner or group of a symbolic link.

EXAMPLES

None.

APPLICATION USAGE

Although *chown*

- 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.

Issue 7

The *fchownat*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 2.

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

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 1003.1-200x defers to the ISO C standard.

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 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

6887 **NAME**

6888 clearerr — clear indicators on a stream

6889 **SYNOPSIS**

6890
```
#include <stdio.h>
```

6891
```
void clearerr(FILE *stream);
```

6892 **DESCRIPTION**

6893 CX The functionality described on this reference page is aligned with the ISO C standard. Any
6894 conflict between the requirements described here and the ISO C standard is unintentional. This
6895 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

6896 The *clearerr*( ) function shall clear the end-of-file and error indicators for the stream to which
6897 *stream* points.

6898 **RETURN VALUE**

6899 The *clearerr*( ) function shall not return a value.

6900 **ERRORS**

6901 No errors are defined.

6902 **EXAMPLES**

6903 None.

6904 **APPLICATION USAGE**

6905 None.

6906 **RATIONALE**

6907 None.

6908 **FUTURE DIRECTIONS**

6909 None.

6910 **SEE ALSO**

6911 The Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

6912 **CHANGE HISTORY**

6913 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

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 1003.1-200x defers to the ISO C standard.

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 XSI 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 1003.1-200x, **<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

```
#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 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

NAME

clock_getres, clock_gettime, clock_settime — clock and timer functions

SYNOPSIS

CX
```
#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.

MON 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.

CPT If _POSIX_CPUTIME is defined, implementations shall support clock ID values obtained by invoking *clock_getcpuclockid*( ), which represent the CPU-time clock of a given process. Implementations shall also support the special **clockid_t** value CLOCK_PROCESS_CPUTIME_ID, which represents the CPU-time clock of the calling process when invoking one of the *clock_**( ) or *timer_**( ) functions. For these clock IDs, the values returned by *clock_gettime*( ) and specified by *clock_settime*( ) represent the amount of execution time of the process associated with the clock. Changing the value of a CPU-time clock via *clock_settime*( ) shall have no effect on the behavior of the sporadic server scheduling policy (see Scheduling Policies (on page 44)).

TCT If _POSIX_THREAD_CPUTIME is defined, implementations shall support clock ID values obtained by invoking *pthread_getcpuclockid*( ), which represent the CPU-time clock of a given thread. Implementations shall also support the special **clockid_t** value CLOCK_THREAD_CPUTIME_ID, which represents the CPU-time clock of the calling thread when invoking one of the *clock_**( ) or *timer_**( ) functions. For these clock IDs, the values returned by *clock_gettime*( ) and specified by *clock_settime*( ) shall represent the amount of execution time of the thread associated with the clock. Changing the value of a CPU-time clock via *clock_settime*( ) shall have no effect on the behavior of the sporadic server scheduling policy (see Scheduling Policies (on page 44)).

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:

[EINVAL] The *clock_id* argument does not specify a known clock.

The *clock_settime*( ) function shall fail if:

[EINVAL] The *tp* argument to *clock_settime*( ) is outside the range for the given clock ID.

[EINVAL] The *tp* argument specified a nanosecond value less than zero or greater than or equal to 1 000 million.

MON [EINVAL] The value of the *clock_id* argument is CLOCK_MONOTONIC.

The *clock_settime*( ) function may fail if:

[EPERM] The requesting process does not have the appropriate privilege to set the specified clock.

EXAMPLES

None.

APPLICATION USAGE

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_getoverrun*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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 IEEE Std 1003.1d-1999.

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 are 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.

Issue 7

Functionality relating to the Clock Selection option is moved to the Base.

The *clock_getres*( ), *clock_gettime*( ), and *clock_settime*( ) functions are moved from the Timers option to the Base.

NAME

clock_nanosleep — high resolution sleep with specifiable clock

SYNOPSIS

CX
```
#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.

7157 **ERRORS**

7158 The *clock_nanosleep*( ) function shall fail if:

7159 [EINTR] The *clock_nanosleep*( ) function was interrupted by a signal.

7160 [EINVAL] The *rqtp* argument specified a nanosecond value less than zero or greater than
7161 or equal to 1 000 million; or the TIMER_ABSTIME flag was specified in flags
7162 and the *rqtp* argument is outside the range for the clock specified by *clock_id*;
7163 or the *clock_id* argument does not specify a known clock, or specifies the CPU-
7164 time clock of the calling thread.

7165 [ENOTSUP] The *clock_id* argument specifies a clock for which *clock_nanosleep*( ) is not
7166 supported, such as a CPU-time clock.

7167 **EXAMPLES**

7168 None.

7169 **APPLICATION USAGE**

7170 Calling *clock_nanosleep*( ) with the value TIMER_ABSTIME not set in the *flags* argument and with
7171 a *clock_id* of CLOCK_REALTIME is equivalent to calling *nanosleep*( ) with the same *rqtp* and *rmtp*
7172 arguments.

7173 **RATIONALE**

7174 The *nanosleep*( ) function specifies that the system-wide clock CLOCK_REALTIME is used to
7175 measure the elapsed time for this time service. However, with the introduction of the monotonic
7176 clock CLOCK_MONOTONIC a new relative sleep function is needed to allow an application to
7177 take advantage of the special characteristics of this clock.

7178 There are many applications in which a process needs to be suspended and then activated
7179 multiple times in a periodic way; for example, to poll the status of a non-interrupting device or
7180 to refresh a display device. For these cases, it is known that precise periodic activation cannot be
7181 achieved with a relative *sleep*( ) or *nanosleep*( ) function call. Suppose, for example, a periodic
7182 process that is activated at time $T0$, executes for a while, and then wants to suspend itself until
7183 time $T0+T$, the period being T. If this process wants to use the *nanosleep*( ) function, it must first
7184 call *clock_gettime*( ) to get the current time, then calculate the difference between the current time
7185 and $T0+T$ and, finally, call *nanosleep*( ) using the computed interval. However, the process could
7186 be preempted by a different process between the two function calls, and in this case the interval
7187 computed would be wrong; the process would wake up later than desired. This problem would
7188 not occur with the absolute *clock_nanosleep*( ) function, since only one function call would be
7189 necessary to suspend the process until the desired time. In other cases, however, a relative sleep
7190 is needed, and that is why both functionalities are required.

7191 Although it is possible to implement periodic processes using the timers interface, this
7192 implementation would require the use of signals, and the reservation of some signal numbers. In
7193 this regard, the reasons for including an absolute version of the *clock_nanosleep*( ) function in
7194 IEEE Std 1003.1-200x are the same as for the inclusion of the relative *nanosleep*( ).

7195 It is also possible to implement precise periodic processes using *pthread_cond_timedwait*( ), in
7196 which an absolute timeout is specified that takes effect if the condition variable involved is never
7197 signaled. However, the use of this interface is unnatural, and involves performing other
7198 operations on mutexes and condition variables that imply an unnecessary overhead.
7199 Furthermore, *pthread_cond_timedwait*( ) is not available in implementations that do not support
7200 threads.

7201 Although the interface of the relative and absolute versions of the new high resolution sleep
7202 service is the same *clock_nanosleep*( ) function, the *rmtp* argument is only used in the relative
7203 sleep. This argument is needed in the relative *clock_nanosleep*( ) function to reissue the function
7204 call if it is interrupted by a signal, but it is not needed in the absolute *clock_nanosleep*( ) function
7205 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 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

Issue 7

The *clock_nanosleep*( ) function is moved from the Clock Selection option to the Base.

7216 **NAME**

7217 clock_settime — clock and timer functions

7218 **SYNOPSIS**

7219 CX
```
#include <time.h>

int clock_settime(clockid_t clock_id, const struct timespec *tp);
```

7221 **DESCRIPTION**

7222 Refer to *clock_getres( )*.

7223 **NAME**

7224 clog, clogf, clogl — complex natural logarithm functions

7225 **SYNOPSIS**

7226
```
#include <complex.h>
```

7227
```
double complex clog(double complex z);
```
7228
```
float complex clogf(float complex z);
```
7229
```
long double complex clogl(long double complex z);
```

7230 **DESCRIPTION**

7231 CX The functionality described on this reference page is aligned with the ISO C standard. Any
7232 conflict between the requirements described here and the ISO C standard is unintentional. This
7233 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

7234 These functions shall compute the complex natural (base *e*) logarithm of *z*, with a branch cut
7235 along the negative real axis.

7236 **RETURN VALUE**

7237 These functions shall return the complex natural logarithm value, in the range of a strip
7238 mathematically unbounded along the real axis and in the interval $[-i\pi, +i\pi]$ along the imaginary
7239 axis.

7240 **ERRORS**

7241 No errors are defined.

7242 **EXAMPLES**

7243 None.

7244 **APPLICATION USAGE**

7245 None.

7246 **RATIONALE**

7247 None.

7248 **FUTURE DIRECTIONS**

7249 None.

7250 **SEE ALSO**

7251 *cexp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

7252 **CHANGE HISTORY**

7253 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

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.

OB 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 controlling process, 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.

OB XSR If *fildes* refers to the slave side of a STREAMS-based pseudo-terminal, a zero-length message may be sent to the master.

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.

SHM If a memory mapped file or a shared memory object 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 SHM memory object becomes unreferenced. If this is the last close of a memory mapped file or a shared memory object 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>

#define LOCKFILE "/etc/ptmp"
...
int pfd;
```

```
FILE *fpfd;
...
if ((fpfd = fdopen (pfd, "w")) == NULL) {
    close(pfd);
    unlink(LOCKFILE);
    exit(1);
}
...
```

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 1003.1-200x only intends to permit such behavior by specifying the [EINTR] error condition.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.6 (on page 38), *exec*, *fattach*( ), *fclose*( ), *fdetach*( ), *fopen*( ), *ioctl*( ), *open*( ), *unlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/12 is applied, correcting the XSH shaded text relating to the master side of a pseudo-terminal. The reason for the change is that the behavior of pseudo-terminals and regular terminals should be as much alike as possible in this case; the change achieves that and matches historical behavior.

Issue 7

Functionality relating to the XSI STREAMS option is marked obsolescent.

Functionality relating to the Asynchronous Input and Output and Memory Mapped Files options is moved to the Base.

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.

[EINTR] 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) {
...
    }

    closedir(dir);
...
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

dirfd( ), *fdopendir*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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

XSI

```
#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*`$`" 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.

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.

ERRORS

No errors are defined.

EXAMPLES

Using openlog( )

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.

```
#include <syslog.h>

char *ident = "Process demo";
int logopt = LOG_PID | LOG_CONS;
int facility = LOG_USER;
...
openlog(ident, logopt, facility);
```

Using setlogmask( )

The following example causes subsequent calls to *syslog*( ) to accept error messages, and to reject all other messages.

```
#include <syslog.h>

int result;
int mask = LOG_MASK (LOG_ERR);
...
result = setlogmask(mask);
```

Using syslog

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.

```
#include <syslog.h>

char *message = "This is a message";
int priority = LOG_ERR | LOG_USER;
...
syslog(priority, message);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

printf( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<syslog.h>**

CHANGE HISTORY

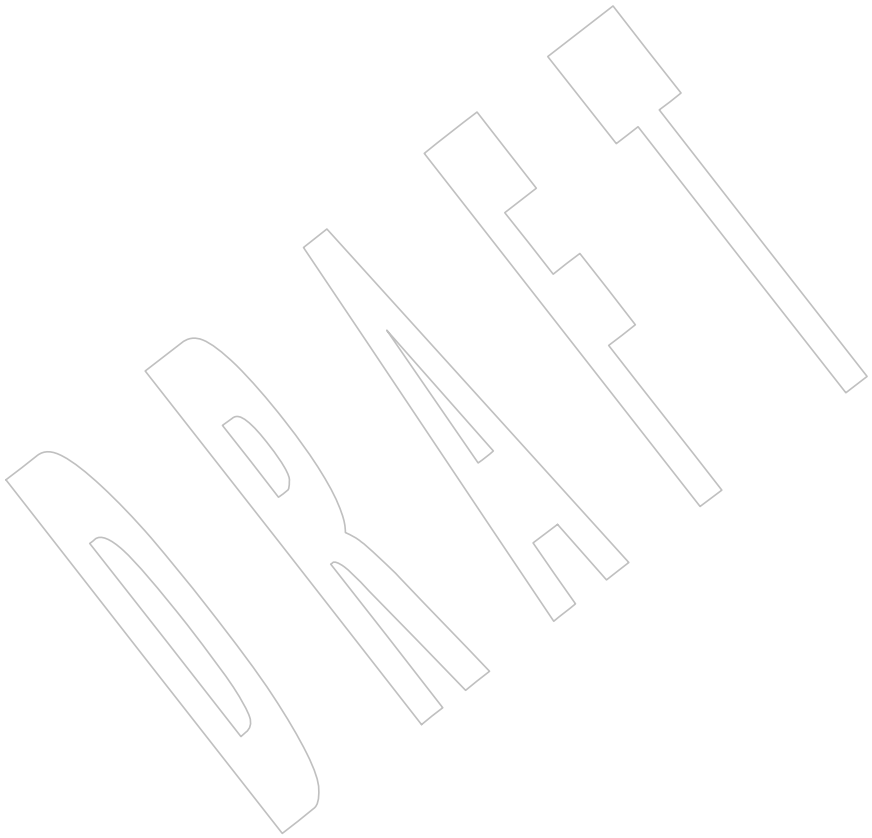
First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/13 is applied, correcting the EXAMPLES section.



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_V7_ILP32_OFF32_CFLAGS
_CS_POSIX_V7_ILP32_OFF32_LDFLAGS
_CS_POSIX_V7_ILP32_OFF32_LIBS
_CS_POSIX_V7_ILP32_OFFBIG_CFLAGS
_CS_POSIX_V7_ILP32_OFFBIG_LDFLAGS
_CS_POSIX_V7_ILP32_OFFBIG_LIBS
_CS_POSIX_V7_LP64_OFF64_CFLAGS
_CS_POSIX_V7_LP64_OFF64_LDFLAGS
_CS_POSIX_V7_LP64_OFF64_LIBS
_CS_POSIX_V7_LPBIG_OFFBIG_CFLAGS
_CS_POSIX_V7_LPBIG_OFFBIG_LDFLAGS
_CS_POSIX_V7_LPBIG_OFFBIG_LIBS
_CS_POSIX_V7_WIDTH_RESTRICTED_ENVS
_CS_V7_ENV

OB

_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
_CS_V6_ENV

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.

After a call to:

```
confstr(_CS_V7_ENV, buf, sizeof(buf))
```

the string stored in *buf* will contain the space-separated list of variable=value environment variable pairs required by the implementation to create a conforming environment, as described in the implementations' conformance documentation.

If the implementation supports the POSIX shell option, the string stored in *buf* after a call to:

```
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 1003.1-200x, 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 1003.1-200x, 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:

```
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 1 024 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:

```
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 1003.1-200x.

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

exec, *pathconf*( ), *sysconf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x, *c99*

CHANGE HISTORY

First released in Issue 4. Derived from the ISO POSIX-2 standard.

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.

Issue 7

Austin Group Interpretation 1003.1-2001 #047 is applied, adding the _CS_V7_ENV variable.

The V6 variables for the supported programming environments are marked obsolescent.

The variables for the supported programming environments are updated to be V7.

The LEGACY variables and obsolescent values are removed.

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

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 1003.1-200x 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 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

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 connection-mode socket or to set or reset the peer address of a connectionless-mode 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. Note that despite no connection being made, the term "connected" is used to describe a connectionless-mode socket for which a peer address has been set.

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.

[EAFNOSUPPORT]
The specified address is not a valid address for the address family of the specified socket.

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

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

[ECONNREFUSED]
The target address was not listening for connections or refused the connection request.

[EINPROGRESS] O_NONBLOCK is set for the file descriptor for the socket and the connection cannot be immediately established; the connection shall be established asynchronously.

[EINTR] The attempt to establish a connection was interrupted by delivery of a signal that was caught; the connection shall be established asynchronously.

[EISCONN] The specified socket is connection-mode and is already connected.

[ENETUNREACH]
No route to the network is present.

[ENOTSOCK] The *socket* argument does not refer to a socket.

[EPROTOTYPE] The specified address has a different type than the socket bound to the specified peer address.

[ETIMEDOUT] The attempt to connect timed out before a connection was made.

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*.

[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 *address* is not a directory.

The *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_len* argument is not a valid length for the address family; or invalid address family in the **sockaddr** structure.

[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 1003.1-200x, **<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.

Issue 7

Austin Group Interpretation 1003.1-2001 #035 is applied, clarifying the description of connected sockets.

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

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 1003.1-200x defers to the ISO C standard.

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

signbit( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

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 1003.1-200x 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 *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 cosine of *x*.

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, 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.

EXAMPLES

Taking the Cosine of a 45-Degree Angle

```
#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 (*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

acos( ), *feclearexcept( )*, *fetestexcept( )*, *isnan( )*, *sin( )*, *tan( )*, the Base Definitions volume of IEEE Std 1003.1-200x, 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.

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

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 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

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 *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 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 1003.1-200x, 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.

7980 **NAME**

7981 cosl — cosine function

7982 **SYNOPSIS**

7983
```
#include <math.h>
```

7984
```
long double cosl(long double x);
```

7985 **DESCRIPTION**

7986 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

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 1003.1-200x 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 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

8017 **NAME**

8018 cproj, cprojf, cprojl — complex projection functions

8019 **SYNOPSIS**

8020
```
#include <complex.h>
```

8021
```
double complex cproj(double complex z);
```
8022
```
float complex cprojf(float complex z);
```
8023
```
long double complex cprojl(long double complex z);
```

8024 **DESCRIPTION**

8025 CX The functionality described on this reference page is aligned with the ISO C standard. Any
8026 conflict between the requirements described here and the ISO C standard is unintentional. This
8027 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

8028 These functions shall compute a projection of *z* onto the Riemann sphere: *z* projects to *z*, except
8029 that all complex infinities (even those with one infinite part and one NaN part) project to
8030 positive infinity on the real axis. If *z* has an infinite part, then *cproj*(*z*) shall be equivalent to:

8031
```
INFINITY + I * copysign(0.0, cimag(z))
```

8032 **RETURN VALUE**

8033 These functions shall return the value of the projection onto the Riemann sphere.

8034 **ERRORS**

8035 No errors are defined.

8036 **EXAMPLES**

8037 None.

8038 **APPLICATION USAGE**

8039 None.

8040 **RATIONALE**

8041 Two topologies are commonly used in complex mathematics: the complex plane with its
8042 continuum of infinities, and the Riemann sphere with its single infinity. The complex plane is
8043 better suited for transcendental functions, the Riemann sphere for algebraic functions. The
8044 complex types with their multiplicity of infinities provide a useful (though imperfect) model for
8045 the complex plane. The *cproj*( ) function helps model the Riemann sphere by mapping all
8046 infinities to one, and should be used just before any operation, especially comparisons, that
8047 might give spurious results for any of the other infinities. Note that a complex value with one
8048 infinite part and one NaN part is regarded as an infinity, not a NaN, because if one part is
8049 infinite, the complex value is infinite independent of the value of the other part. For the same
8050 reason, *cabs*( ) returns an infinity if its argument has an infinite part and a NaN part.

8051 **FUTURE DIRECTIONS**

8052 None.

8053 **SEE ALSO**

8054 *carg*( ), *cimag*( ), *conj*( ), *creal*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
8055 **<complex.h>**

8056 **CHANGE HISTORY**

8057 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

8058 **NAME**

8059 creal, crealf, creall — complex real functions

8060 **SYNOPSIS**

8061
```
#include <complex.h>
```

8062
```
double creal(double complex z);
```
8063
```
float crealf(float complex z);
```
8064
```
long double creall(long double complex z);
```

8065 **DESCRIPTION**

8066 CX The functionality described on this reference page is aligned with the ISO C standard. Any
8067 conflict between the requirements described here and the ISO C standard is unintentional. This
8068 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

8069 These functions shall compute the real part of *z*.

8070 **RETURN VALUE**

8071 These functions shall return the real part value.

8072 **ERRORS**

8073 No errors are defined.

8074 **EXAMPLES**

8075 None.

8076 **APPLICATION USAGE**

8077 For a variable *z* of type **complex**:

8078
```
z == creal(z) + cimag(z)*I
```

8079 **RATIONALE**

8080 None.

8081 **FUTURE DIRECTIONS**

8082 None.

8083 **SEE ALSO**

8084 *carg( )*, *cimag( )*, *conj( )*, *cproj( )*, the Base Definitions volume of IEEE Std 1003.1-200x,
8085 **<complex.h>**

8086 **CHANGE HISTORY**

8087 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

NAME

creat — create a new file or rewrite an existing one

SYNOPSIS

OH
```
#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

mknod( ), *open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

8137 **NAME**

8138 crypt — string encoding function (**CRYPT**)

8139 **SYNOPSIS**

8140 XSI
```
#include <unistd.h>
```
8141
```
char *crypt(const char *key, const char *salt);
```

8142 **DESCRIPTION**

8143 The *crypt*( ) function is a string encoding function. The algorithm is implementation-defined.

8144 The *key* argument points to a string to be encoded. The *salt* argument is a string chosen from the
8145 set:

8146
```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```
8147
```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```
8148
```
0 1 2 3 4 5 6 7 8 9 . /
```

8149 The first two characters of this string may be used to perturb the encoding algorithm.

8150 The return value of *crypt*( ) points to static data that is overwritten by each call.

8151 The *crypt*( ) function need not be thread-safe. A function that is not required to be thread-safe is
8152 not required to be reentrant.

8153 **RETURN VALUE**

8154 Upon successful completion, *crypt*( ) shall return a pointer to the encoded string. The first two
8155 characters of the returned value shall be those of the *salt* argument. Otherwise, it shall return a
8156 null pointer and set *errno* to indicate the error.

8157 **ERRORS**

8158 The *crypt*( ) function shall fail if:

8159 [ENOSYS] The functionality is not supported on this implementation.

8160 **EXAMPLES**

8161 **Encoding Passwords**

8162 The following example finds a user database entry matching a particular user name and changes
8163 the current password to a new password. The *crypt*( ) function generates an encoded version of
8164 each password. The first call to *crypt*( ) produces an encoded version of the old password; that
8165 encoded password is then compared to the password stored in the user database. The second
8166 call to *crypt*( ) encodes the new password before it is stored.

8167 The *putpwent*( ) function, used in the following example, is not part of IEEE Std 1003.1-200x.

```
8168 #include <unistd.h>
8169 #include <pwd.h>
8170 #include <string.h>
8171 #include <stdio.h>
8172 ...
8173 int valid_change;
8174 int pfd;  /* Integer for file descriptor returned by open(). */
8175 FILE *fpfd;  /* File pointer for use in putpwent(). */
8176 struct passwd *p;
8177 char user[100];
8178 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 1003.1-200x, **<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

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 1003.1-200x 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.

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 1003.1-200x 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 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

8267 **NAME**

8268 csinl — complex sine functions

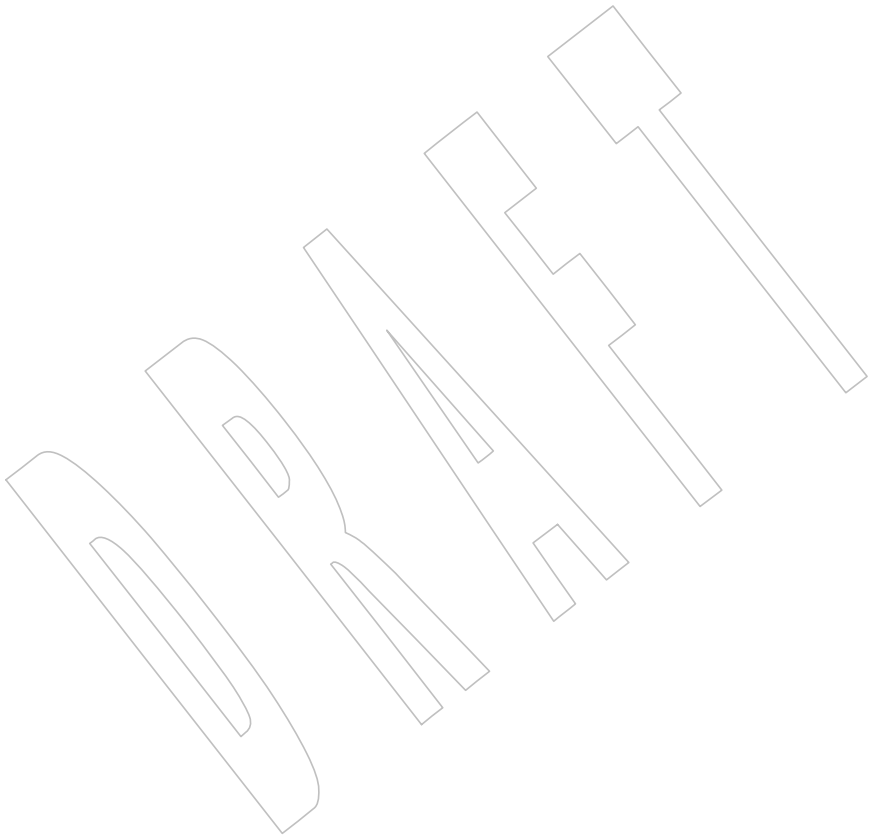
8269 **SYNOPSIS**

8270
```
#include <complex.h>
```

8271
```
long double complex csinl(long double complex z);
```

8272 **DESCRIPTION**

8273 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

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 1003.1-200x 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.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

cabs( ), *cpow*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

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

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 1003.1-200x 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.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

catan( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

NAME

ctanh, 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

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 1003.1-200x defers to the ISO C standard.

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

catanh( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

8360 **NAME**

8361 ctanl — complex tangent functions

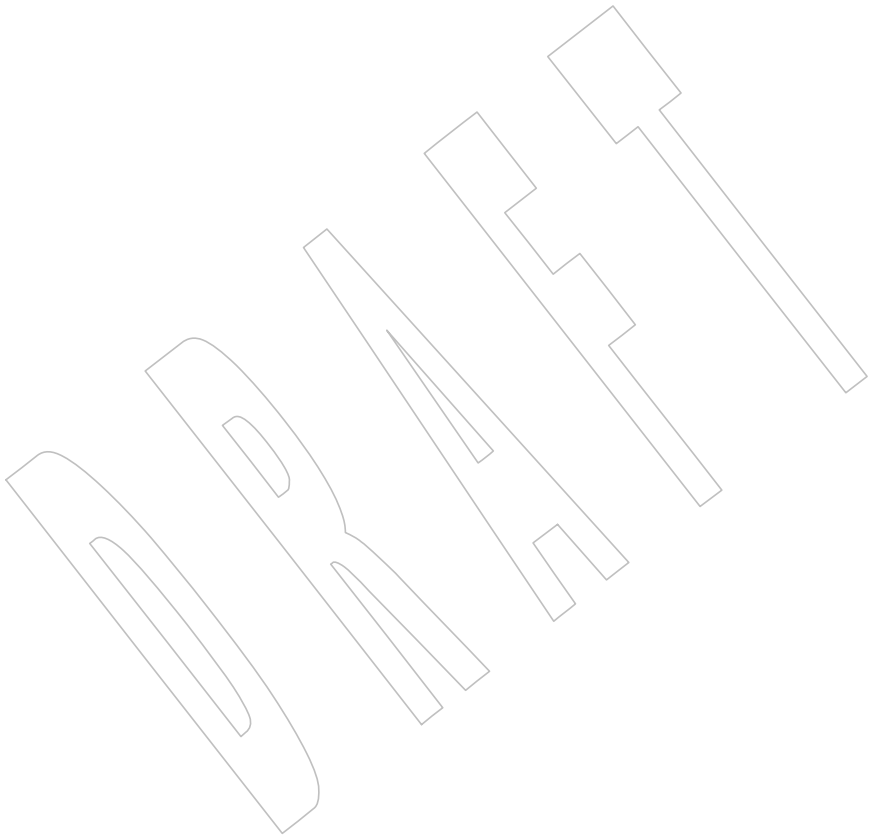
8362 **SYNOPSIS**

8363
```
#include <complex.h>
```

8364
```
long double complex ctanl(long double complex z);
```

8365 **DESCRIPTION**

8366 Refer to *ctan( )*.



NAME

ctermid — generate a pathname for the controlling terminal

SYNOPSIS

CX
```
#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 1003.1-200x, **<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 normative text 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

OB
```
#include <time.h>

char *ctime(const time_t *clock);
```
OB CX
```
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 1003.1-200x 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 thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

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.

CX 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

These functions are included only for compatibility with older implementations. They have undefined behavior if the resulting string would be too long, so the use of these functions should be discouraged. On implementations that do not detect output string length overflow, it is possible to overflow the output buffers in such a way as to cause applications to fail, or possible system security violations. Also, these functions do not support localized date and time formats. To avoid these problems, applications should use *strftime*( ) to generate strings from broken-down times.

Values for the broken-down time structure can be obtained by calling *gmtime*( ) or *localtime*( ).

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.

Attempts to use *ctime*( ) or *ctime_r*( ) for times before the Epoch or for times beyond the year 9999 produce undefined results. Refer to *asctime*( ) (on page 130).

RATIONALE

The standards developers decided to mark the *ctime*( ) and *ctime_r*( ) functions obsolescent even though they are in the ISO C standard due to the possibility of buffer overflow. The ISO C standard also provides the *strftime*( ) function which can be used to avoid these problems.

FUTURE DIRECTIONS

These functions may be removed in a future version.

SEE ALSO

asctime( ), *clock*( ), *difftime*( ), *gmtime*( ), *localtime*( ), *mktime*( ), *strftime*( ), *strptime*( ), *time*( ), *utime*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

Issue 7

SD5-XSH-ERN-25 is applied, updating the APPLICATION USAGE.

Austin Group Interpretation 1003.1-2001 #053 is applied, marking these functions obsolescent.

The *ctime_r*( ) function is moved from the Thread-Safe Functions option to the Base.

8496 **NAME**

8497 daylight — daylight savings time flag

8498 **SYNOPSIS**

8499 XSI
```
#include <time.h>

extern int daylight;
```

8501 **DESCRIPTION**

8502 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

XSI

```
#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*.

A database shall be stored in one or two files. When one file is used, the name of the database file shall be formed by appending the suffix **.db** to the *file* argument given to *dbm_open*( ). When two files are used, the names of the database files shall be formed by appending the suffixes **.dir** and **.pag** respectively to the *file* argument.

The *dbm_open*( ) function shall open a database. The *file* argument to the function is the pathname of the database. 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_open*( ) function need not accept pathnames longer than {PATH_MAX}−4 bytes (including the terminating null), or pathnames with a last component longer than {NAME_MAX}−4 bytes (excluding the terminating null).

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 thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

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.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The following code can be used to traverse the database:

```
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.

Applications should take care that database pathname arguments specified to *dbm_open*( ) are not prefixes of unrelated files. This might be done, for example, by placing databases in a separate directory.

Since some implementations use three characters for a suffix and others use four characters for a suffix, applications should ensure that the maximum portable pathname length passed to *dbm_open*( ) is no greater than {PATH_MAX}–4 bytes, with the last component of the pathname no greater than {NAME_MAX}–4 bytes.

RATIONALE

Previously the standard required the database to be stored in two files, one file being a directory containing a bitmap of keys and having **.dir** as its suffix. The second file containing all data and having **.pag** as its suffix. This has been changed not to specify the use of the files and to allow newer implementations of the Berkeley DB interface using a single file that have evolved while remaining compatible with the application programming interface. The standard developers considered removing the specific suffixes altogether but decided to retain them so as not to pollute the application file name space more than necessary and to allow for portable backups of the database.

FUTURE DIRECTIONS

None.

SEE ALSO

open( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

Issue 6

The normative text is updated to avoid use of the term "must" for application requirements.

Issue 7

Austin Group Interpretation 1003.1-2001 #042 is applied so that the DESCRIPTION permits newer implementations of the Berkeley DB interface.

difftime( )

NAME

difftime — compute the difference between two calendar time values

SYNOPSIS

```
#include <time.h>

double difftime(time_t time1, time_t time0);
```

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 1003.1-200x defers to the ISO C standard.

The *difftime*( ) function shall compute the differ

8682 **NAME**

8683 dirfd — extract the file descriptor used by a DIR stream

8684 **SYNOPSIS**

8685
```
#include <dirent.h>

int dirfd(DIR *dirp);
```

8687 **DESCRIPTION**

8688 The *dirfd*( ) function shall return a file descriptor referring to the same directory as the *dirp*
8689 argument. This file descriptor shall be closed by a call to *closedir*( ). The behavior of future calls
8690 to *readdir*( ) and *readdir_r*( ) is undefined if the application attempts to alter the file position
8691 indicator using the returned file descriptor. The behavior of future calls to *closedir*( ), *readdir*( ),
8692 and *readdir_r*( ) is undefined if the application attempts to close the file descriptor.

8693 **RETURN VALUE**

8694 Upon successful completion, the *dirfd*( ) function shall return an integer which contains a file
8695 descriptor for the stream pointed to by *dirp*. Otherwise, it shall return –1 and may set *errno* to
8696 indicate the error.

8697 **ERRORS**

8698 The *dirfd*( ) function may fail if:

8699 [EINVAL] The *dirp* argument does not refer to a valid directory stream.

8700 [ENOTSUP] The implementation does not support the association of a file descriptor with
8701 a directory.

8702 **EXAMPLES**

8703 None.

8704 **APPLICATION USAGE**

8705 The *dirfd*( ) function is intended to be a mechanism by which an application may obtain a file
8706 descriptor to use for the *fchdir*( ) function.

8707 **RATIONALE**

8708 This interface was introduced because the Base Definitions volume of IEEE Std 1003.1-200x does
8709 not make public the **DIR** data structure. Applications tend to use the *fchdir*( ) function on the file
8710 descriptor returned by this interface, and this has proven useful for security reasons; in
8711 particular, it is a better technique than others where directory names might change.

8712 The description uses the term "a file descriptor" rather than "the file descriptor". The
8713 implication intended is that an implementation that does not use an *fd* for *diropen*( ) could still
8714 *open*( ) the directory to implement the *dirfd*( ) function. Such a descriptor must be closed later
8715 during a call to *closedir*( ).

8716 An implementation that does not support file descriptors referring to directories may fail with
8717 [ENOTSUP].

8718 If it is necessary to allocate an *fd* to be returned by *dirfd*( ), it should be done at the time of a call
8719 to *opendir*( ).

8720 **FUTURE DIRECTIONS**

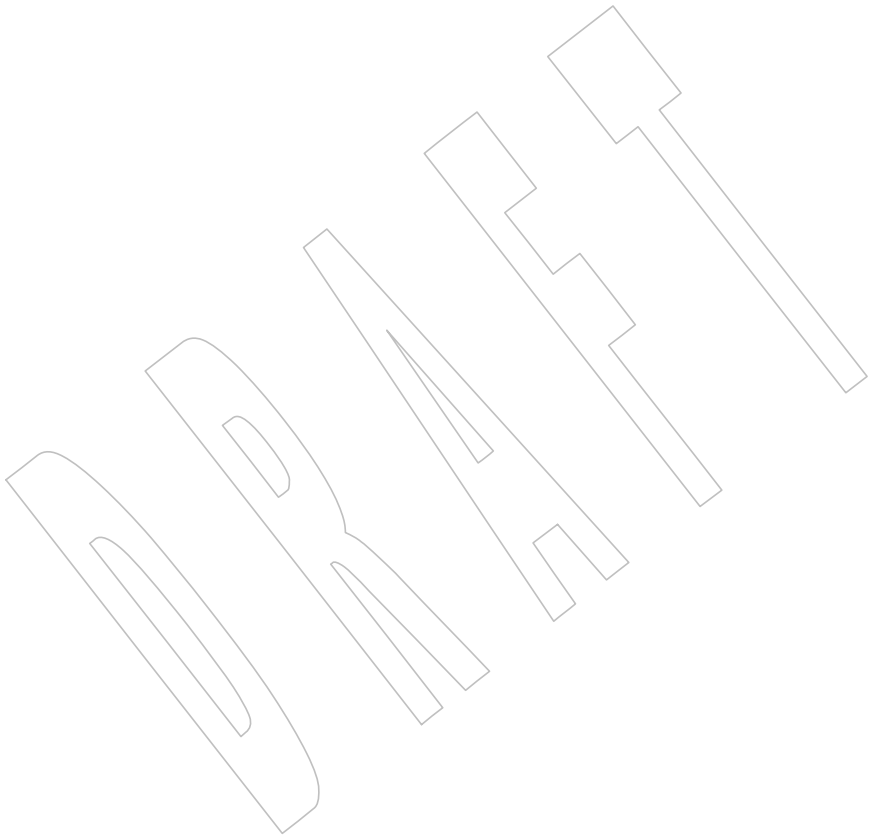
8721 None.

SEE ALSO

closedir( ), *fchdir( )*, *fdopendir( )*, *fileno( )*, *open( )*, *opendir( )*, *readdir( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<dirent.h>**

CHANGE HISTORY

First released in Issue 7.



NAME

dirname — report the parent directory name of a file pathname

SYNOPSIS

XSI
```
#include <libgen.h>

char *dirname(char *path);
```

DESCRIPTION

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 thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

RETURN VALUE

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*( ).

ERRORS

No errors are defined.

EXAMPLES

The following code fragment reads a pathname, changes the current working directory to the parent directory, and opens the file.

```
char path[PATH_MAX], *pathcopy;
int fd;
fgets(path, PATH_MAX, stdin);
pathcopy = strdup(path);
chdir(dirname(pathcopy));
fd = open(basename(path), O_RDONLY);
```

Sample Input and Output Strings for dirname( )

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.

Input String	Output String
"/usr/lib"	"/usr"
"/usr/"	"/"
"usr"	"."
"/"	"/"
"."	"."
".."	"."

Changing the Current Directory to the Parent Directory

The following program fragment reads a pathname, changes the current working directory to the parent directory, and opens the file.

```
#include <unistd.h>
#include <limits.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <libgen.h>
...
char path[PATH_MAX], *pathcopy;
int fd;
...
fgets(path, PATH_MAX, stdin);
pathcopy = strdup(path);
chdir(dirname(pathcopy));
fd = open(basename(path), O_RDONLY);
```

APPLICATION USAGE

The *dirname*( ) and *basename*( ) functions together yield a complete pathname. The expression *dirname*(*path*) obtains the pathname of the directory where *basename*(*path*) is found.

Since the meaning of the leading `"//"` is implementation-defined, *dirname*("**//foo**) may return either `"//"` or `'//'` (but nothing else).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

basename( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

8801 **NAME**

8802 div — compute the quotient and remainder of an integer division

8803 **SYNOPSIS**

8804
```
#include <stdlib.h>
```

8805
```
div_t div(int numer, int denom);
```

8806 **DESCRIPTION**

8807 CX The functionality described on this reference page is aligned with the ISO C standard. Any
8808 conflict between the requirements described here and the ISO C standard is unintentional. This
8809 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

8810 The *div*( ) function shall compute the quotient and remainder of the division of the numerator
8811 *numer* by the denominator *denom*. If the division is inexact, the resulting quotient is the integer
8812 of lesser magnitude that is the nearest to the algebraic quotient. If the result cannot be
8813 represented, the behavior is undefined; otherwise, *quot*denom+rem* shall equal *numer*.

8814 **RETURN VALUE**

8815 The *div*( ) function shall return a structure of type **div_t**, comprising both the quotient and the
8816 remainder. The structure includes the following members, in any order:

8817
```
int  quot;  /* quotient */
```
8818
```
int  rem;   /* remainder */
```

8819 **ERRORS**

8820 No errors are defined.

8821 **EXAMPLES**

8822 None.

8823 **APPLICATION USAGE**

8824 None.

8825 **RATIONALE**

8826 None.

8827 **FUTURE DIRECTIONS**

8828 None.

8829 **SEE ALSO**

8830 *ldiv*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

8831 **CHANGE HISTORY**

8832 First released in Issue 4. Derived from the ISO C standard.

8833 **NAME**

8834 dlclose — close a *dlopen*( ) object

8835 **SYNOPSIS**

8836
```
#include <dlfcn.h>
```

8837
```
int dlclose(void *handle);
```

8838 **DESCRIPTION**

8839 The *dlclose*( ) function shall inform the system that the object referenced by a *handle* returned
8840 from a previous *dlopen*( ) invocation is no longer needed by the application.

8841 The use of *dlclose*( ) reflects a statement of intent on the part of the process, but does not create
8842 any requirement upon the implementation, such as removal of the code or symbols referenced
8843 by *handle*. Once an object has been closed using *dlclose*( ) an application should assume that its
8844 symbols are no longer available to *dlsym*( ). All objects loaded automatically as a result of
8845 invoking *dlopen*( ) on the referenced object shall also be closed if this is the last reference to it.

8846 Although a *dlclose*( ) operation is not required to remove structures from an address space,
8847 neither is an implementation prohibited from doing so. The only restriction on such a removal is
8848 that no object shall be removed to which references have been relocated, until or unless all such
8849 references are removed. For instance, an object that had been loaded with a *dlopen*( ) operation
8850 specifying the RTLD_GLOBAL flag might provide a target for dynamic relocations performed in
8851 the processing of other objects—in such environments, an application may assume that no
8852 relocation, once made, shall be undone or remade unless the object requiring the relocation has
8853 itself been removed.

8854 **RETURN VALUE**

8855 If the referenced object was successfully closed, *dlclose*( ) shall return 0. If the object could not be
8856 closed, or if *handle* does not refer to an open object, *dlclose*( ) shall return a non-zero value. More
8857 detailed diagnostic information shall be available through *dlerror*( ).

8858 **ERRORS**

8859 No errors are defined.

8860 **EXAMPLES**

8861 The following example illustrates use of *dlopen*( ) and *dlclose*( ):

8862
```
...
```

8863
```
/* Open a dynamic library and then close it ... */
```

8864
```
#include <dlfcn.h>
```

8865
```
void *mylib;
```

8866
```
int eret;
```

8867
```
mylib = dlopen("mylib.so", RTLD_LOCAL | RTLD_LAZY);
```

8868
```
...
```

8869
```
eret = dlclose(mylib);
```

8870
```
...
```

8871 **APPLICATION USAGE**

8872 A conforming application should employ a *handle* returned from a *dlopen*( ) invocation only
8873 within a given scope bracketed by the *dlopen*( ) and *dlclose*( ) operations. Implementations are
8874 free to use reference counting or other techniques such that multiple calls to *dlopen*( ) referencing
8875 the same object may return the same object for *handle*. Implementations are also free to reuse a
8876 *handle*. For these reasons, the value of a *handle* must be treated as an opaque object by the
8877 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 1003.1-200x, **<dlfcn.h>**

CHANGE HISTORY

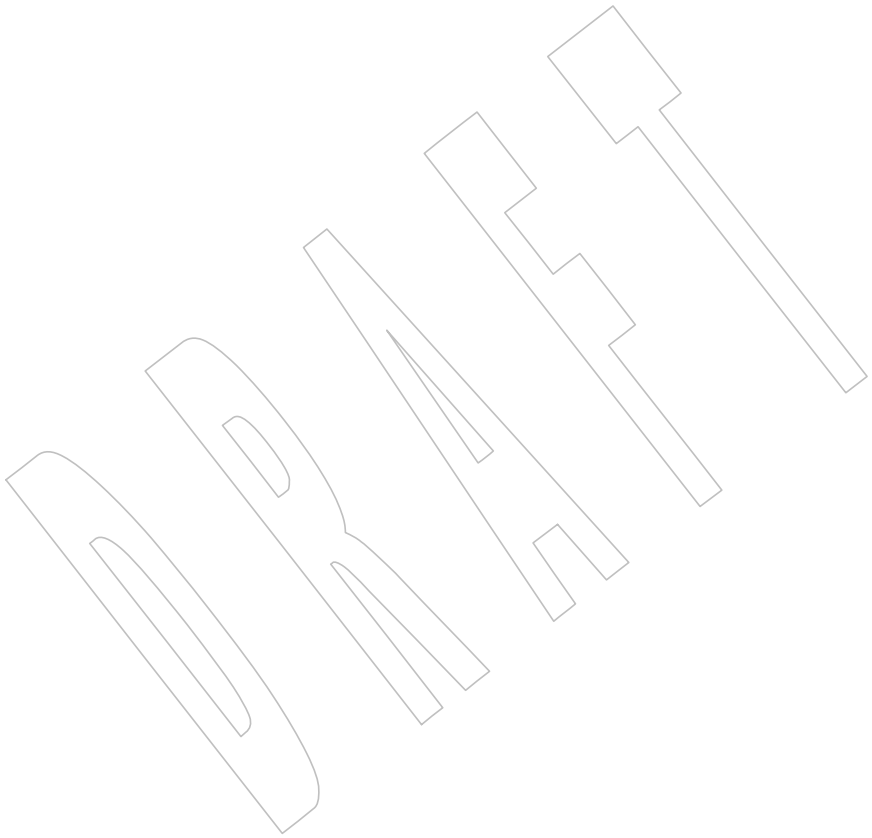
First released in Issue 5.

Issue 6

The DESCRIPTION is updated to say that the referenced object is closed "if this is the last reference to it".

Issue 7

The *dlopen( )* function is moved from the XSI option to Base.



NAME

dlerror — get diagnostic information

SYNOPSIS

```
#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 thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

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 1003.1-200x, **<dlfcn.h>**

CHANGE HISTORY

First released in Issue 5.

Issue 6

In the DESCRIPTION the note about reentrancy and thread-safety is added.

Issue 7

The *dlerror*( ) function is moved from the XSI option to the Base.

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 *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 *dlopen*( ) invocation, and any objects that were loaded in any *dlopen*( ) invocation and which specified the RTLD_GLOBAL flag. To determine the scope of visibility for the symbols loaded with a *dlopen*( ) invocation, the *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 *dlopen*(*0, mode*) and an associated *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 the default behavior is unspecified.

If a *file* is specified in multiple *dlopen*( ) invocations, *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 *dlclose*( )).

Symbols introduced into a program through calls to *dlopen*( ) may be used in relocation activities. Symbols so introduced may duplicate symbols already defined by the program or previous *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 *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 *dlopen*( ) operation on a *file* of 0, dependency ordering is used by the *dlsym*( ) function. Load ordering is used in *dlsym*( ) operations upon the global symbol object.

When an object is first made accessible via *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 *dlopen*( ) operations and available through *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 *extern* linkage. The precise manner in which an implementation constructs the set of exported symbols for a *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 *dlopen*( ), or if an error occurs during the process of loading *file* or relocating its symbolic references, *dlopen*( ) shall return NULL. More detailed diagnostic information shall be available through *dlerror*( ).

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 1003.1-200x, **<dlfcn.h>**

CHANGE HISTORY

First released in Issue 5.

Issue 6

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/21 is applied, changing the default behavior in the DESCRIPTION when neither RTLD_GLOBAL nor RTLD_LOCAL are specified from implementation-defined to unspecified.

Issue 7

The *dlopen*( ) function is moved from the XSI option to the Base.

NAME

dlsym — obtain the address of a symbol from a *dlopen*( ) object

SYNOPSIS

```
#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.

```
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 */
*(void **)(&fptr) = 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

a *dlopen*( ) operation including 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

The ISO C standard does not require that pointers to functions can be cast back and forth to pointers to data. However, POSIX-conforming implementations are required to support his, as noted in Section 2.12.3 (on page 83). The result of converting a pointer to a function into a pointer to another data type (except **void** *) is still undefined, however.

Note that compilers conforming to the ISO C standard are required to generate a warning if a conversion from a **void** * pointer to a function pointer is attempted as in:

```
fptr = (int (*)(int))dlsym(handle, "my_function");
```

FUTURE DIRECTIONS

None.

SEE ALSO

dlclose( ), *dlerror*( ), *dlopen*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/14 is applied, correcting an example, and adding text to the RATIONALE describing issues related to conversion of pointers to functions and back again.

Issue 7

The *dlsym*( ) function is moved from the XSI option to the Base.

NAME

dprintf — print formatted output

SYNOPSIS

CX

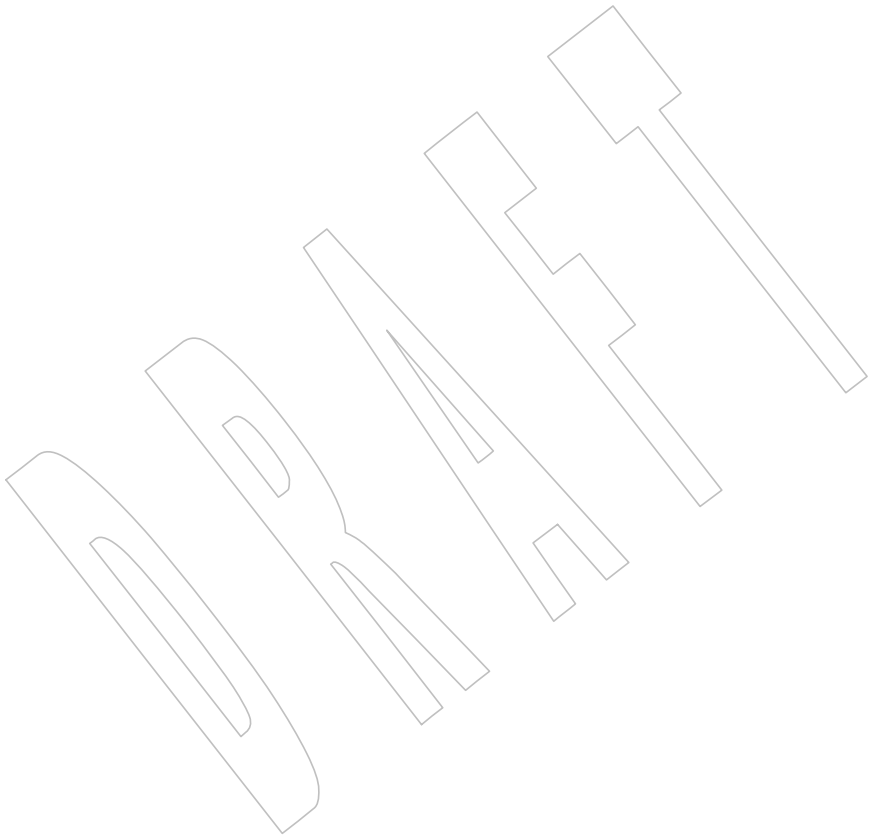
```
#include <stdio.h>

int dprintf(int fildes, const char *format, ...);
```

DESCRIPTION

Refer to *fprintf( )*.

NAME

drand48, erand48, jrand48, lcong48, lrand48, mrand48, nrand48, seed48, srand48 — generate



provide storage for the successive X_i values in the array specified as an argument when the functions are invoked. That is why these routines do not have to be initialized; the calling program merely has to place the desired initial value of X_i into the array and pass it as an argument. By using different arguments, *erand48*( ), *nrand48*( ), and *jrand48*( ) allow separate modules of a large program to generate several *independent* streams of pseudo-random numbers; that is, the sequence of numbers in each stream shall *not* depend upon how many times the routines are called to generate numbers for the other streams.

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

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 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

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

Issue 5

A note indicating that the *drand48*( ), *lrand48*( ), and *mrand48*( ) functions need not be reentrant is added to the DESCRIPTION.

Issue 6

The normative text 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] All file descriptors available to the process are currently open.

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.

```
#include <unistd.h>
...
int pfd;
...
close(1);
dup(pfd);
close(pfd);
...
```

Redirecting Error Messages

The following example redirects messages from *stderr* to *stdout*.

```
#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 1003.1-200x 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 1003.1-200x 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 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

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

NAME

duplocale — duplicate a locale object

SYNOPSIS

CX
```
#include <locale.h>

locale_t duplocale(locale_t locobj);
```

DESCRIPTION

The *duplocale*( ) function shall create a duplicate copy of the locale object referenced by the *locobj* argument.

RETURN VALUE

Upon successful completion, the *duplocale*( ) function shall return a handle for a new locale object. Otherwise, *duplocale*( ) shall return (**locale_t**)0 and set *errno* to indicate the error.

ERRORS

The *duplocale*( ) function shall fail if:

[ENOMEM] There is not enough memory available to create the locale object or load the locale data.

The *duplocale*( ) function may fail if:

[EINVAL] *locobj* is not a handle for a locale object.

EXAMPLES

Constructing an Altered Version of an Existing Locale Object

The following example shows a code fragment to create a slightly altered version of an existing locale object. The function takes a locale object and a locale name and it replaces the *LC_TIME* category data in the locale object with that from the named locale.

```
#include <locale.h>
...

locale_t
with_changed_lc_time (locale_t obj, const char *name)
{
    locale_t retval = duplocale (obj);
    if (retval != (locale_t) 0)
    {
        locale_t changed = newlocale (LC_TIME_MASK, name, retval);
        if (changed == (locale_t) 0)
            /* An error occurred. Free all allocated resources. */
            freelocale (retval);
        retval = changed;
    }
    return retval; }
}
```

APPLICATION USAGE

The use of the *duplocale*( ) function is recommended for situations where a locale object is being used in multiple places, and it is possible that the lifetime of the locale object might end before all uses are finished. Another reason to duplicate a locale object is if a slightly modified form is needed. This can be achieved by a call to *newlocale*( ) following the *duplocale*( ) call.

As with the *newlocale*( ) function, handles for locale objects created by the *duplocale*( ) function should be released by a corresponding call to *freelocale*( ).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

freelocale( ), *newlocale*( ), *uselocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<locale.h>**

CHANGE HISTORY

First released in Issue 7.

NAME

encrypt — encoding function (**CRYPT**)

SYNOPSIS

XSI

```
#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 thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

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 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

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

Issue 5

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

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 thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

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:

[EINTR] A signal was caught during the operation.

[EIO] An I/O error has occurred.

[EMFILE] All file descriptors available to the process are currently open.

[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 1003.1-200x, **<grp.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 RETURN VALUE section.

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.

Issue 7

SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

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*( ), 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.

The *endhostent*( ) function shall close the connection to the database, releasing any open file descriptor.

These functions need not be thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

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( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<netdb.h>**

9495 **CHANGE HISTORY**

9496 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

endnetent, getnetbyaddr, getnetbyname, getnetent, setnetent — network database functions

SYNOPSIS

```
#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 thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

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 1003.1-200x, **<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 thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

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 1003.1-200x, **<netdb.h>**

CHANGE HISTORY

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

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 thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

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] All file descriptors available to the process are currently open.

[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*( ).

EXAMPLES

Searching the User Database

The following example uses the *getpwent*( ) function to get successive entries in the user database, returning a pointer to a **passwd** structure that contains information about each user. The call to *endpwent*( ) closes the user database and cleans up.

```
#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 1003.1-200x, **<pwd.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 RETURN VALUE section.

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.

Issue 7

SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

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 *getserv**( ) 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 a value obtained by converting a **uint16_t** in network byte order to **int**.

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 thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

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 1003.1-200x, **<netdb.h>**

CHANGE HISTORY

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

Issue 7

SD5-XBD-ERN-14 is applied.

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:

ut_type Member	Other Members with Meaningful Data
EMPTY	No others
BOOT_TIME	*ut_tv*
OLD_TIME	*ut_tv*
NEW_TIME	*ut_tv*
USER_PROCESS	*ut_id*, *ut_user* (login name of the user), *ut_line*, *ut_pid*, *ut_tv*
INIT_PROCESS	*ut_id*, *ut_pid*, *ut_tv*
LOGIN_PROCESS	*ut_id*, *ut_user* (implementation-defined name of the login process), *ut_pid*, *ut_tv*
DEAD_PROCESS	*ut_id*, *ut_pid*, *ut_tv*

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 thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

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.

EXAMPLES

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 1003.1-200x, **<utmpx.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.

9827 **NAME**

9828 environ — array of character pointers to the environment strings

9829 **SYNOPSIS**

9830
```
extern char **environ;
```

9831 **DESCRIPTION**

9832 Refer to the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables
9833 and *exec*.

NAME

erand48 — generate uniformly distributed pseudo-random numbers

SYNOPSIS

XSI

```
#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

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 1003.1-200x defers to the ISO C standard.

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 *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 value of the error function.

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

If *x* is ±0, ±0 s

```
}
```

APPLICATION USAGE

Underflow occurs when $|x| < \text{DBL\_MIN} * (sqrt(\pi)/2)$.

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 1003.1-200x, 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 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.

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

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/22 is applied, adding the example to the EXAMPLES section.

9909 **NAME**

9910 erfc, erfcf, erfcl — complementary error functions

9911 **SYNOPSIS**

9912
```
#include <math.h>

double erfc(double x);
float erfcf(float x);
long double erfcl(long double x);
```

9916 **DESCRIPTION**

9917 CX The functionality described on this reference page is aligned with the ISO C standard. Any
9918 conflict between the requirements described here and the ISO C standard is unintentional. This
9919 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

9920 These functions shall compute the complementary error function 1.0 – *erf* (*x*).

9921 An application wishing to check for error situations should set *errno* to zero and call
9922 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
9923 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
9924 zero, an error has occurred.

9925 **RETURN VALUE**

9926 Upon successful completion, these functions shall return the value of the complementary error
9927 function.

9928 If the correct value would cause underflow and is not representable, a range error may occur
9929 MX and either 0.0 (if representable), or an implementation-defined value shall be returned.

9930 MX If *x* is NaN, a NaN shall be returned.

9931 If *x* is ±0, +1 shall be returned.

9932 If *x* is –Inf, +2 shall be returned.

9933 If *x* is +Inf, +0 shall be returned.

9934 If the correct value would cause underflow and is representable, a range error may occur and
9935 the correct value shall be returned.

9936 **ERRORS**

9937 These functions may fail if:

9938 Range Error The result underflows.

9939 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
9940 then *errno* shall be set to [ERANGE]. If the integer expression
9941 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
9942 floating-point exception shall be raised.

9943 **EXAMPLES**

9944 None.

9945 **APPLICATION USAGE**

9946 The *erfc*( ) function is provided because of the extreme loss of relative accuracy if *erf* (*x*) is called
9947 for large *x* and the result subtracted from 1.0.

9948 Note for IEEE Std 754-1985 **double**, 26.55 < *x* implies *erfc*(*x*) has underflowed.

9949 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
9950 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 1003.1-200x, 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 *erfc( )* 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.

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

9970 **NAME**

9971 erff, erfl — error functions

9972 **SYNOPSIS**

9973
```
#include <math.h>
```

9974
```
float erff(float x);
```

9975
```
long double erfl(long double x);
```

9976 **DESCRIPTION**

9977 Refer to *erf*( ).

NAME

errno — error return value

SYNOPSIS

```
#include <errno.h>
```

DESCRIPTION

The lvalue *errno* is used by many functions to return error values.

Many functions provide an error number in *errno*, which has type **int** and is defined in **<errno.h>**. The value of *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 *errno* should only be examined when it is indicated to be valid by a function's return value. Applications shall obtain the definition of *errno* by the inclusion of **<errno.h>**. No function in this volume of IEEE Std 1003.1-200x shall set *errno* to 0. The setting of *errno* after a successful call to a function is unspecified unless the description of that function specifies that *errno* shall not be modified.

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 symbolic values stored in *errno* are documented in the ERRORS sections on all relevant pages.

RETURN VALUE

None.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

Previously both POSIX and X/Open documents were more restrictive than the ISO C standard in that they required *errno* to be defined as an external variable, whereas the ISO C standard required only that *errno* be defined as a modifiable lvalue with type **int**.

An application that needs to examine the value of *errno* to determine the error should set it to 0 before a function call, then inspect it before a subsequent function call.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.3, the Base Definitions volume of IEEE Std 1003.1-200x, **<errno.h>**

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 *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:

```
extern int errno;
```


Issue 6

Obsolescent text regarding defining *errno* as:

```
extern int errno
```

is removed.

Text regarding no function setting *errno* to zero to indicate an error is changed to no function shall set *errno* to zero. This is for alignment with the ISO/IEC 9899: 1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/23 is applied, adding text to the DESCRIPTION stating that the setting of *errno* after a successful call to a function is unspecified unless the description of the function requires that it will not be modified.

10031 **NAME**

10032 environ, execl, execv, execle, execve, execlp, execvp, fexecve — execute a file

10033 **SYNOPSIS**

10034
```
#include <unistd.h>
```

10035
```
extern char **environ;
```
10036
```
int execl(const char *path, const char *arg0, ... /*, (char *)0 */);
```
10037
```
int execv(const char *path, char *const argv[]);
```
10038
```
int execle(const char *path, const char *arg0, ... /*,
```
10039
```
    (char *)0, char *const envp[]*/);
```
10040
```
int execve(const char *path, char *const argv[], char *const envp[]);
```
10041
```
int execlp(const char *file, const char *arg0, ... /*, (char *)0 */);
```
10042
```
int execvp(const char *file, char *const argv[]);
```
10043
```
int fexecve(int fd, char *const argv[], char *const envp[]);
```

10044 **DESCRIPTION**

10045 The *exec* family of functions shall replace the current process image with a new process image.
10046 The new image shall be constructed from a regular, executable file called the *new process image*
10047 *file*. There shall be no return from a successful *exec*, because the calling process image is overlaid
10048 by the new process image.

10049 The *fexecve*() function shall be equivalent to the *execve*() function except that the file to be
10050 executed is determined by the file descriptor *fd* instead of a pathname. The file offset of *fd* is
10051 ignored.

10052 When a C-language program is executed as a result of a call to one of the *exec* family of
10053 functions, it shall be entered as a C-language function call as follows:

10054
```
int main (int argc, char *argv[]);
```

10055 where *argc* is the argument count and *argv* is an array of character pointers to the arguments
10056 themselves. In addition, the following variable:

10057
```
extern char **environ;
```

10058 is initialized as a pointer to an array of character pointers to the environment strings. The *argv*
10059 and *environ* arrays are each terminated by a null pointer. The null pointer terminating the *argv*
10060 array is not counted in *argc*.

10061 Conforming multi-threaded applications shall not use the *environ* variable to access or modify
10062 any environment variable while any other thread is concurrently modifying any environment
10063 variable. A call to any function dependent on any environment variable shall be considered a
10064 use of the *environ* variable to access that environment variable.

10065 The arguments specified by a program with one of the *exec* functions shall be passed on to the
10066 new process image in the corresponding *main*() arguments.

10067 The argument *path* points to a pathname that identifies the new process image file.

10068 The argument *file* is used to construct a pathname that identifies the new process image file. If
10069 the *file* argument contains a slash character, the *file* argument shall be used as the pathname for
10070 this file. Otherwise, the path prefix for this file is obtained by a search of the directories passed
10071 as the environment variable *PATH* (see the Base Definitions volume of IEEE Std 1003.1-200x,
10072 Chapter 8, Environment Variables). If this environment variable is not present, the results of the
10073 search are implementation-defined.

10074 There are two distinct ways in which the contents of the process image file may cause the
10075 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:

```
execl(<shell path>, arg0, file, arg1, ..., (char *)0);
```

where <*shell path*> is an unspecified pathname for the *sh* utility, *file* is the process image file, and for *execvp*(), where *arg*0, *arg*1, and so on correspond to the values passed to *execvp*() in *argv*[0], *argv*[1], and so on.

The arguments represented by *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 *arg0* should point to a filename that is associated with the process being started by one of the *exec* functions.

The argument *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 *argv*[0] should point to a filename that is associated with the process being started by one of the *exec* functions.

The argument *envp* is an array of character pointers to null-terminated strings. These strings shall constitute the environment for the new process image. The *envp* array is terminated by a null pointer.

For those forms not containing an *envp* pointer (*execl*(), *execv*(), *execlp*(), and *execvp*()), the environment for the new process image shall be taken from the external variable *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 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 *close*(). Locks that are not removed by closing of file descriptors remain unchanged.

If file descriptors 0, 1, and 2 would otherwise be closed after a successful call to one of the *exec* family of functions, and the new process image file has the set-user-ID or set-group-ID file mode
XSI bits set, and the ST_NOSUID bit is not set for the file system containing the new process image file, implementations may open an unspecified file for each of these file descriptors in the new process image.

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 initial thread of the new process image shall be set to the default.

The state of conversion descriptors and message catalog descriptors in the new process image is undefined.

For the new process image, the equivalent of:

```
setlocale(LC_ALL, "C")
```

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 **<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.

XSI After a successful call to any of the *exec* functions, alternate signal stacks are not preserved and the SA_ONSTACK flag shall be cleared for all signals.

After a successful call to any of the *exec* functions, any functions previously registered by the *atexit*( ) or *pthread_atfork*( ) functions 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.

Any named semaphores open in the calling process shall be closed as if by appropriate calls to *sem_close*( ).

TYM 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.

Memory mappings created in the process are unmapped before the address space is rebuilt for the new process image.

When the calling process image does not use the SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC scheduling policies, the scheduling policy and parameters of the new process image and the initial thread in that new process image are implementation-defined.

PS When the calling process image uses the SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC scheduling policies, the process policy and scheduling parameter settings shall not be changed

TPS by a call to an *exec* function. The initial thread in the new process image shall inherit the process scheduling policy and parameters. It shall have the default system contention scope, but shall inherit its allocation domain from the calling process image.

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*( ).

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.

OB 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 thread ID of the initial thread in the new process image is unspecified.

The size and location of the stack on which the initial thread in the new process image runs is unspecified.

The initial thread in the new process image shall have its cancellation type set to PTHREAD_CANCEL_DEFERRED and its cancellation state set to PTHREAD_CANCEL_ENABLED.

The initial thread in the new process image shall have all thread-specific data values set to NULL and all thread-specific data keys shall be removed by the call to *exec* without running destructors.

The initial thread in the new process image shall be joinable, as if created with the *detachstate* attribute set to PTHREAD_CREATE_JOINABLE.

The new process shall inherit at least the following attributes from the calling process image:

XSI • Nice value (see *nice*())

XSI • *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*())

10210 XSI
- File size limit (see *ulimit*( ))

10211
- Process signal mask (see *sigprocmask*( ))

10212
- Pending signal (see *sigpending*( ))

10213
- *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* (see *times*( ))

10214 XSI
- Resource limits

10215
- Controlling terminal

10216 XSI
- Interval timers

10217 10218 The initial thread of the new process shall inherit at least the following attributes from the calling thread:

10219
- Signal mask (see *sigprocmask*( ) and *pthread_sigmask*( ))

10220
- Pending signals (see *sigpending*( ))

10221 10222 10223 10224 10225 All other process attributes defined in this volume of IEEE Std 1003.1-200x shall be inherited in the new process image from the old process image. All other thread attributes defined in this volume of IEEE Std 1003.1-200x shall be inherited in the initial thread in the new process image from the calling thread in the old process image. The inheritance of process or thread attributes not defined by this volume of IEEE Std 1003.1-200x is implementation-defined.

10226 10227 10228 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 or cleanup handlers shall be called.

10229 10230 10231 10232 10233 10234 10235 10236 10237 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.

10238 XSI 10239 The saved resource limits in the new process image are set to be a copy of the process' corresponding hard and soft limits.

10240 **RETURN VALUE**

10241 10242 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.

10243 **ERRORS**

10244 The *exec* functions shall fail if:

10245 10246 10247 [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.

10248 10249 10250 10251 [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.

10252 10253 10254 [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 *execlp*( ) and *execvp*( ), shall fail if:

[ENOEXEC] The new process image file has the appropriate access permission but has an unrecognized format.

The *fexecve*( ) function shall fail if:

[EBADF] The *fd* argument is not a valid file descriptor open for executing.

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 execl( )

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.

```
#include <unistd.h>

int ret;
...
ret = execl ("/bin/ls", "ls", "-1", (char *)0);
```

Using execle( )

The following example is similar to Using execl( ) (on page 312). In addition, it specifies the environment for the new process image using the *env* argument.

```
#include <unistd.h>

int ret;
char *env[] = { "HOME=/usr/home", "LOGNAME=home", (char *)0 };
...
ret = execle ("/bin/ls", "ls", "-l", (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.

```
#include <unistd.h>

int ret;
...
ret = execlp ("ls", "ls", "-l", (char *)0);
```

Using execv()

The following example passes arguments to the *ls* command in the *cmd* array.

```
#include <unistd.h>

int ret;
char *cmd[] = { "ls", "-l", (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.

```
#include <unistd.h>

int ret;
char *cmd[] = { "ls", "-l", (char *)0 };
char *env[] = { "HOME=/usr/home", "LOGNAME=home", (char *)0 };
...
ret = execve ("/bin/ls", cmd, env);
```

Using execvp()

The following example searches for the location of the *ls* command among the directories specified by the *PATH* environment variable, and passes arguments to the *ls* command in the *cmd* array.

```
#include <unistd.h>

int ret;
char *cmd[] = { "ls", "-l", (char *)0 };
...
ret = execvp ("ls", cmd);
```

APPLICATION USAGE

As the state of conversion descriptors and message catalog descriptors in the new process image is undefined, conforming applications should not rely on their use and should close them prior to calling one of the *exec* functions.

Applications that require other than the default POSIX locale should call *setlocale*() with the appropriate parameters to establish the locale of the new process.

The *environ* array should not be accessed directly by the application.

The new process might be invoked in a non-conforming environment if the *envp* array does not contain implementation-defined variables required by the implementation to provide a conforming environment. See the _CS_V7_ENV entry in **<unistd.h>** and *confstr*() for details.

Applications should not depend on file descriptors 0, 1, and 2 being closed after an *exec*. A future version may allow these file descriptors to be automatically opened for any process.

If an application wants to perform a checksum test of the file being executed before executing it, the file will need to be opened with read permission to perform the checksum test.

Since execute permission is checked by *fexecve*(), the file description *fd* need not have been opened with the O_EXEC flag. However, if the file to be executed denies read and write permission for the process preparing to do the *exec*, the only way to provide the *fd* to *fexecve*() will be to use the O_EXEC flag when opening *fd*. In this case, the application will not be able to perform a checksum test since it will not be able to read the contents of the file.

Note that when a file descriptor is opened with O_RDONLY, O_RDWR, or O_WRONLY mode, the file descriptor can be used to read, read and write, or write the file, respectively, even if the mode of the file changes after the file was opened. Using the O_EXEC open mode is different; *fexecve*() will ignore the mode that was used when the file descriptor was opened and the *exec* will fail if the mode of the file associated with *fd* does not grant execute permission to the calling process at the time *fexecve*() is called.

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 1003.1-200x as well. The wording, in particular the use of the word *should*, requires a Strictly Conforming POSIX Application to pass at least one argument to the *exec* function, thus guaranteeing that *argc* be one or greater when invoked by such an application. In fact, this is good practice, since many existing applications reference *argv*[0] without first checking the value of *argc*.

The requirement on a Strictly Conforming POSIX Application also states that the value passed as the first argument be a filename associated with the process being started. Although some existing applications pass a pathname rather than a filename in some circumstances, a filename is more generally useful, since the common usage of *argv*[0] is in printing diagnostics. In some cases the filename passed is not the actual filename of the file; for example, many implementations of the *login* utility use a convention of prefixing a hyphen ('-') to the actual filename, which indicates to the command interpreter being invoked that it is a "login shell".

Historically there have been two ways that implementations can *exec* shell scripts.

One common historical implementation is that the *execl*(), *execv*(), *execle*(), and *execve*() functions return an [ENOEXEC] error for any file not recognizable as executable, including a shell script. When the *execlp*() and *execvp*() functions encounter such a file, they assume the file to be a shell script and invoke a known command interpreter to interpret such files. This is now required by IEEE Std 1003.1-200x. These implementations of *execvp*() and *execlp*() only give the [ENOEXEC] error in the rare case of a problem with the command interpreter's executable file. Because of these implementations, the [ENOEXEC] error is not mentioned for *execlp*() or *execvp*(), although implementations can still give it.

Another way that some historical implementations handle shell scripts is by recognizing the first two bytes of the file as the character string "#!" and using the remainder of the first line of the file as the name of the command interpreter to execute.

One potential source of confusion noted by the standard developers is over how the contents of a process image file affect the behavior of the *exec* family of functions. The following is a description of the actions taken:

1. If the process image file is a valid executable (in a format that is executable and valid and having appropriate permission) for this system, then the system executes the file.
2. If the process image file has appropriate permission and is in a format that is executable but not valid for this system (such as a recognized binary for another architecture), then this is an error and *errno* is set to [EINVAL] (see later RATIONALE on [EINVAL]).
3. If the process image file has appropriate permission but is not otherwise recognized:
   a. If this is a call to *execlp*( ) or *execvp*( ), then they invoke a command interpreter assuming that the process image file is a shell script.
   b. If this is not a call to *execlp*( ) or *execvp*( ), then an error occurs and *errno* is set to [ENOEXEC].

Applications that do not require to access their arguments may use the form:

```
main(void)
```

as specified in the ISO C standard. However, the implementation will always provide the two arguments *argc* and *argv*, even if they are not used.

Some implementations provide a third argument to *main*( ) called *envp*. This is defined as a pointer to the environment. The ISO C standard specifies invoking *main*( ) with two arguments, so implementations must support applications written this way. Since this volume of IEEE Std 1003.1-200x 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 1003.1-200x specifies that signals set to SIG_IGN remain set to SIG_IGN, and that the new process image inherits the signal mask of the thread that called *exec* in the old process image. 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 *exec*s without explicit reason to do so, and especially not to block signals across *exec*s 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:

dst:	**char *[ ]**	**const char *[ ]**	**char *const[ ]**	**const char *const[ ]**
src:				
char *[ ]	VALID	—	VALID	—
const char *[ ]	—	VALID	—	VALID
char * const [ ]	—	—	VALID	—
const char *const[ ]	—	—	—	VALID

Since all existing code has a source type matching the first row, the column that gives the most valid combinations is the third column. The only other possibility is the fourth column, but using it would require a cast on the *argv* or *envp* arguments. It is unfortunate that the fourth column cannot be used, because the declaration a non-expert would naturally use would be that in the second row.

The ISO C standard and this volume of IEEE Std 1003.1-200x 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 1003.1-200x 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 1003.1-200x 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 1003.1-200x 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 1003.1-200x, 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.

The purpose of the *fexecve*( ) function is to enable executing a file which has been verified to be the intended file. It is possible to actively check the file by reading from the file descriptor and be sure that the file is not exchanged for another between the reading and the execution. Alternatively, an function like *openat*( ) can be used to open a file which has been found by reading the content of a directory using *readdir*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

alarm( ), *atexit*( ), *chmod*( ), *close*( ), *confstr*( ), *exit*( ), *fcntl*( ), *fork*( ), *fstatvfs*( ), *getenv*( ), *getitimer*( ), *getrlimit*( ), *mknod*( ), *mmap*( ), *nice*( ), *open*( ), *posix_spawn*( ), *posix_trace_create*( ), *posix_trace_event*( ), *posix_trace_eventid_equal*( ), *pthread_atfork*( ), *pthread_sigmask*( ), *putenv*( ), *readdir*( ), *semop*( ), *setlocale*( ), *shmat*( ), *sigaction*( ), *sigaltstack*( ), *sigpending*( ), *system*( ), *times*( ), *ulimit*( ), *umask*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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 normative text 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.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/15 is applied, adding a new paragraph to the DESCRIPTION and text to the end of the APPLICATION USAGE section. This change addresses a security concern, where implementations may want to reopen file descriptors 0, 1, and 2 for programs with the set-user-id or set-group-id file mode bits calling the *exec* family of functions.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/24 is applied, applying changes to the DESCRIPTION, addressing which attributes are inherited by threads, and behavioral requirements for threads attributes.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/25 is applied, updating text in the RATIONALE from "the process signal mask be unchanged across an *exec*" to "the new process image inherits the signal mask of the thread that called *exec* in the old process image".

Issue 7

Austin Group Interpretation 1003.1-2001 #047 is applied, adding the description of _CS_V7_ENV to the APPLICATION USAGE.

The *fexecve*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 2.

Functionality relating to the Asynchronous Input and Output, Memory Mapped Files, Threads, and Timers options is moved to the Base.

NAME

exit — terminate a process

SYNOPSIS

```
#include <stdlib.h>

void exit(int status);
```

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 1003.1-200x defers to the ISO C standard.

CX The value of *status* may be 0, EXIT_SUCCESS, EXIT_FAILURE, or any other value, though only the least significant 8 bits (that is, *status* & 0377) shall be available to a waiting parent process.

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 and close all CX open streams. Finally, the process shall be terminated with the same consequences as described in Consequences of Process Termination (on page 87).

RETURN VALUE

The *exit*( ) function does not return.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

See *_Exit*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

_Exit( ), *atexit*( ), *exec*, *longjmp*( ), *tmpfile*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

Issue 7

Austin Group Interpretation 1003.1-2001 #031 is applied, separating the *_Exit*( ) and *_exit*( ) functions from the *exit*( ) function.

Austin Group Interpretation 1003.1-2001 #085 is applied.

10586 **NAME**

10587 exp, expf, expl — exponential function

10588 **SYNOPSIS**

10589
```
#include <math.h>
```

10590
```
double exp(double x);
```
10591
```
float expf(float x);
```
10592
```
long double expl(long double x);
```

10593 **DESCRIPTION**

10594 CX The functionality described on this reference page is aligned with the ISO C standard. Any
10595 conflict between the requirements described here and the ISO C standard is unintentional. This
10596 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

10597 These functions shall compute the base-*e* exponential of *x*.

10598 An application wishing to check for error situations should set *errno* to zero and call
10599 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
10600 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
10601 zero, an error has occurred.

10602 **RETURN VALUE**

10603 Upon successful completion, these functions shall return the exponential value of *x*.

10604 If the correct value would cause overflow, a range error shall occur and *exp*( ), *expf*( ), and *expl*( )
10605 shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

10606 If the correct value would cause underflow, and is not representable, a range error may occur,
10607 MX and either 0.0 (if supported), or an implementation-defined value shall be returned.

10608 MX If *x* is NaN, a NaN shall be returned.

10609 If *x* is ±0, 1 shall be returned.

10610 If *x* is –Inf, +0 shall be returned.

10611 If *x* is +Inf, *x* shall be returned.

10612 If the correct value would cause underflow, and is representable, a range error may occur and
10613 the correct value shall be returned.

10614 **ERRORS**

10615 These functions shall fail if:

10616 Range Error The result overflows.

10617 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
10618 then *errno* shall be set to [ERANGE]. If the integer expression
10619 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
10620 floating-point exception shall be raised.

10621 These functions may fail if:

10622 Range Error The result underflows.

10623 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
10624 then *errno* shall be set to [ERANGE]. If the integer expression
10625 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
10626 floating-point exception shall be raised.

EXAMPLES

Computing the Density of the Standard Normal Distribution

This function shows an implementation for the density of the standard normal distribution using *exp*( ). This example uses the constant M_PI which is part of the XSI option.

```
#include <math.h>

double
normal_density (double  x)
{
    return exp(–x*x/2) / sqrt (2*M_PI);
}
```

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_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*( ), the Base Definitions volume of IEEE Std 1003.1-200x, 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.

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

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/26 is applied, adding the example to the EXAMPLES section.

10662 **NAME**

10663 exp2, exp2f, exp2l — exponential base 2 functions

10664 **SYNOPSIS**

10665
```
#include <math.h>

double exp2(double x);
float exp2f(float x);
long double exp2l(long double x);
```

10669 **DESCRIPTION**

10670 CX The functionality described on this reference page is aligned with the ISO C standard. Any
10671 conflict between the requirements described here and the ISO C standard is unintentional. This
10672 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

10673 These functions shall compute the base-2 exponential of *x*.

10674 An application wishing to check for error situations should set *errno* to zero and call
10675 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
10676 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
10677 zero, an error has occurred.

10678 **RETURN VALUE**

10679 Upon successful completion, these functions shall return 2^x.

10680 If the correct value would cause overflow, a range error shall occur and *exp2*( ), *exp2f*( ), and
10681 *exp2l*( ) shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL,
10682 respectively.

10683 If the correct value would cause underflow, and is not representable, a range error may occur,
10684 MX and either 0.0 (if supported), or an implementation-defined value shall be returned.

10685 MX If *x* is NaN, a NaN shall be returned.

10686 If *x* is ±0, 1 shall be returned.

10687 If *x* is –Inf, +0 shall be returned.

10688 If *x* is +Inf, *x* shall be returned.

10689 If the correct value would cause underflow, and is representable, a range error may occur and
10690 the correct value shall be returned.

10691 **ERRORS**

10692 These functions shall fail if:

10693 Range Error The result overflows.

10694 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
10695 then *errno* shall be set to [ERANGE]. If the integer expression
10696 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
10697 floating-point exception shall be raised.

10698 These functions may fail if:

10699 Range Error The result underflows.

10700 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
10701 then *errno* shall be set to [ERANGE]. If the integer expression
10702 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
10703 floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

For IEEE Std 754-1985 **double**, 1024 <= *x* implies *exp2*(*x*) has overflowed. The value *x*< –1022 implies *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

exp( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), *log*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

10721 **NAME**

10722 expm1, expm1f, expm1l — compute exponential functions

10723 **SYNOPSIS**

10724
```
#include <math.h>
```

10725
```
double expm1(double x);
```
10726
```
float expm1f(float x);
```
10727
```
long double expm1l(long double x);
```

10728 **DESCRIPTION**

10729 CX The functionality described on this reference page is aligned with the ISO C standard. Any
10730 conflict between the requirements described here and the ISO C standard is unintentional. This
10731 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

10732 These functions shall compute e^x–1.0.

10733 An application wishing to check for error situations should set *errno* to zero and call
10734 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
10735 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
10736 zero, an error has occurred.

10737 **RETURN VALUE**

10738 Upon successful completion, these functions return e^x–1.0.

10739 If the correct value would cause overflow, a range error shall occur and *expm1*( ), *expm1f*( ), and
10740 *expm1l*( ) shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL,
10741 respectively.

10742 MX If *x* is NaN, a NaN shall be returned.

10743 If *x* is ±0, ±0 shall be returned.

10744 If *x* is –Inf, –1 shall be returned.

10745 If *x* is +Inf, *x* shall be returned.

10746 If *x* is subnormal, a range error may occur and *x* should be returned.

10747 **ERRORS**

10748 These functions shall fail if:

10749 Range Error The result overflows.

10750 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
10751 then *errno* shall be set to [ERANGE]. If the integer expression
10752 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
10753 floating-point exception shall be raised.

10754 These functions may fail if:

10755 MX Range Error The value of *x* is subnormal.

10756 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
10757 then *errno* shall be set to [ERANGE]. If the integer expression
10758 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
10759 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 * 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_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*( ), *ilogb*( ), *log1p*( ), the Base Definitions volume of IEEE Std 1003.1-200x, 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.

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

10791 **NAME**

10792 fabs, fabsf, fabsl — absolute value function

10793 **SYNOPSIS**

10794
```
#include <math.h>

double fabs(double x);
float fabsf(float x);
long double fabsl(long double x);
```

10798 **DESCRIPTION**

10799 CX The functionality described on this reference page is aligned with the ISO C standard. Any
10800 conflict between the requirements described here and the ISO C standard is unintentional. This
10801 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

10802 These functions shall compute the absolute value of their argument x, $|x|$.

10803 **RETURN VALUE**

10804 Upon successful completion, these functions shall return the absolute value of x.

10805 MX If x is NaN, a NaN shall be returned.

10806 If x is ±0, +0 shall be returned.

10807 If x is ±Inf, +Inf shall be returned.

10808 **ERRORS**

10809 No errors are defined.

10810 **EXAMPLES**

10811 **Computing the 1-Norm of a Floating-Point Vector**

10812 This example shows the use of *fabs*( ) to compute the 1-norm of a vector defined as follows:

10813
```
norm1(v) = |v[0]| + |v[1]| + ... + |v[n–1]|
```

10814 where $|x|$ denotes the absolute value of x, n denotes the vector's dimension $v[i]$ denotes the i-th
10815 component of v ($0 \le i < n$).

10816
```
#include <math.h>

double
norm1(const double v[], const int n)
{
    int     i;
    double  n1_v;  /* 1-norm of v */

    n1_v = 0;
    for (i=0; i<n; i++) {
        n1_v += fabs (v[i]);
    }

    return n1_v;
}
```

10828 **APPLICATION USAGE**

10829 None.

10830 **RATIONALE**
10831 None.

10832 **FUTURE DIRECTIONS**
10833 None.

10834 **SEE ALSO**
10835 *isnan( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

10836 **CHANGE HISTORY**
10837 First released in Issue 1. Derived from Issue 1 of the SVID.

10838 **Issue 5**
10839 The DESCRIPTION is updated to indicate how an application should check for an error. This
10840 text was previously published in the APPLICATION USAGE section.

10841 **Issue 6**
10842 The *fabsf( )* and *fabsl( )* functions are added for alignment with the ISO/IEC 9899: 1999 standard.

10843 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
10844 revised to align with the ISO/IEC 9899: 1999 standard.

10845 IEC 60559: 1989 standard floating-point extensions over the ISO/IEC 9899: 1999 standard are
10846 marked.

10847 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/27 is applied, adding the example to the
10848 EXAMPLES section.

faccessat( )

10849 **NAME**

10850 faccessat — determine accessibility of a file relative to directory file descriptor

10851 **SYNOPSIS**

10852 ```
#include <unistd.h>
```

10853 ```
int faccessat(int fd, const char *path, int amode, int flag);
```

10854 **DESCRIPTION**

10855 Refer to *access( )*.

NAME

fattach — attach a STREAMS-based file descriptor to a file in the file system name space (**STREAMS**)

SYNOPSIS

OB 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*.

```
#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

The *fattach*( ) function may be removed in a future version.

SEE ALSO

fdetach( ), *isastream*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

Issue 6

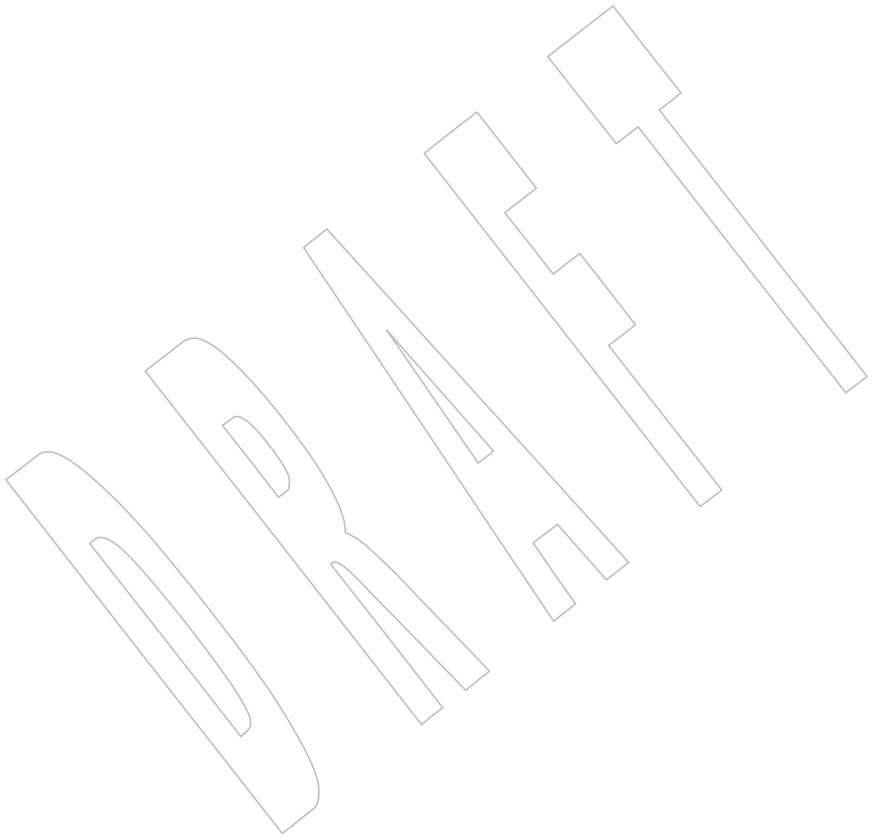
This function is marked as part of the XSI STREAMS Option Group.

The normative text 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.

Issue 7

The *fattach*( ) function is marked obsolescent.



10946 **NAME**

10947 fchdir — change working directory

10948 **SYNOPSIS**

10949 ```
#include <unistd.h>
```

10950 ```
int fchdir(int fildes);
```

10951 **DESCRIPTION**

10952 The *fchdir*( ) function shall be equivalent to *chdir*( ) except that the directory that is to be the new
10953 current working directory is specified by the file descriptor *fildes*.

10954 A conforming application can obtain a file descriptor for a file of type directory using *open*( ),
10955 provided that the file status flags and access modes do not contain O_WRONLY or O_RDWR.

10956 **RETURN VALUE**

10957 Upon successful completion, *fchdir*( ) shall return 0. Otherwise, it shall return –1 and set *errno* to
10958 indicate the error. On failure the current working directory shall remain unchanged.

10959 **ERRORS**

10960 The *fchdir*( ) function shall fail if:

10961 [EACCES] Search permission is denied for the directory referenced by *fildes*.

10962 [EBADF] The *fildes* argument is not an open file descriptor.

10963 [ENOTDIR] The open file descriptor *fildes* does not refer to a directory.

10964 The *fchdir*( ) may fail if:

10965 [EINTR] A signal was caught during the execution of *fchdir*( ).

10966 [EIO] An I/O error occurred while reading from or writing to the file system.

10967 **EXAMPLES**

10968 None.

10969 **APPLICATION USAGE**

10970 None.

10971 **RATIONALE**

10972 None.

10973 **FUTURE DIRECTIONS**

10974 None.

10975 **SEE ALSO**

10976 *chdir*( ), *dirfd*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

10977 **CHANGE HISTORY**

10978 First released in Issue 4, Version 2.

10979 **Issue 5**

10980 Moved from X/OPEN UNIX extension to BASE.

10981 **Issue 7**

10982 The *fchdir*( ) function is moved from the XSI option to the Base.

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.

If *fildes* refers to a socket, the behavior of *fchmod*( ) is unspecified.

OB 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.

[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);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod( ), *chown*( ), *creat*( ), *fcntl*( ), *fstatat*( ), *fstatvfs*( ), *mknod*( ), *open*( ), *read*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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

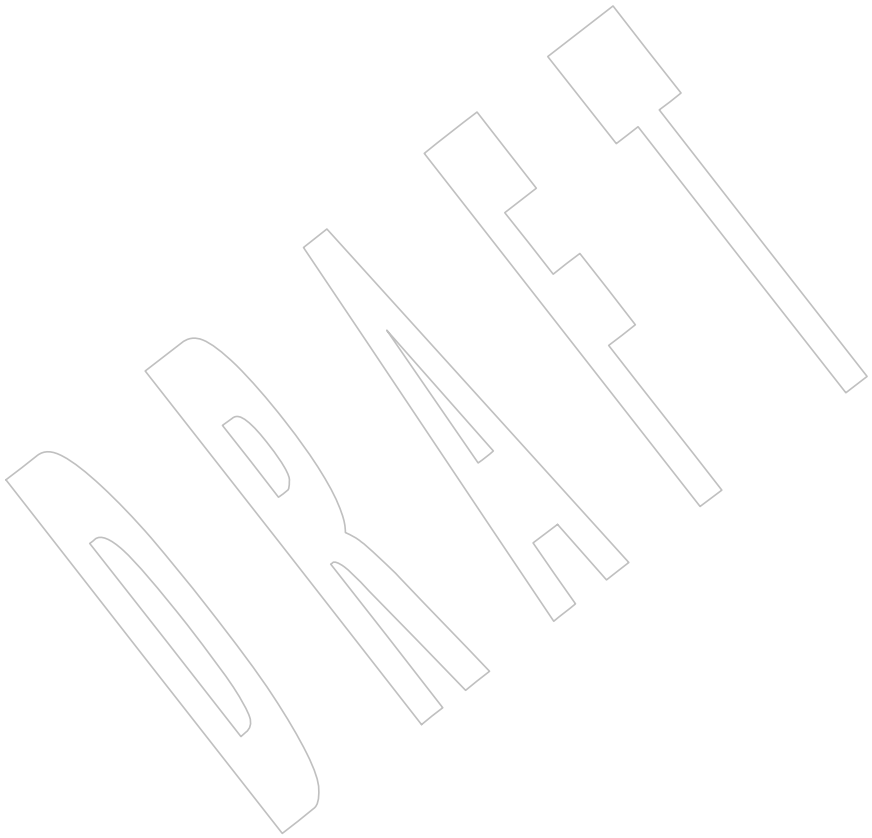
fchmodat — change mode of a file relative to directory file descriptor

SYNOPSIS

```
#include <sys/stat.h>

int fchmodat(int fd, const char *path, mode_t mode, int flag);
```

DESCRIPTION

Refer to *chmod( )*.,i



11048 **NAME**

11049 fchown — change owner and group of a file

11050 **SYNOPSIS**

11051
```
#include <unistd.h>
```

11052
```
int fchown(int fildes, uid_t owner, gid_t group);
```

11053 **DESCRIPTION**

11054 The *fchown*( ) function shall be equivalent to *chown*( ) except that the file whose owner and group
11055 are changed is specified by the file descriptor *fildes*.

11056 **RETURN VALUE**

11057 Upon successful completion, *fchown*( ) shall return 0. Otherwise, it shall return –1 and set *errno* to
11058 indicate the error.

11059 **ERRORS**

11060 The *fchown*( ) function shall fail if:

11061 [EBADF] The *fildes* argument is not an open file descriptor.

11062 [EPERM] The effective user ID does not match the owner of the file or the process does
11063 not have appropriate privilege and _POSIX_CHOWN_RESTRICTED indicates
11064 that such privilege is required.

11065 [EROFS] The file referred to by *fildes* resides on a read-only file system.

11066 The *fchown*( ) function may fail if:

11067 [EINVAL] The owner or group ID is not a value supported by the implementation. The
11068 OB XSR *fildes* argument refers to a pipe or socket or an *fattach*( )-ed STREAM and the
11069 implementation disallows execution of *fchown*( ) on a pipe.

11070 [EIO] A physical I/O error has occurred.

11071 [EINTR] The *fchown*( ) function was interrupted by a signal which was caught.

11072 **EXAMPLES**

11073 **Changing the Current Owner of a File**

11074 The following example shows how to change the owner of a file named **/home/cnd/mod1** to
11075 "jones" and the group to "cnd".

11076 The numeric value for the user ID is obtained by extracting the user ID from the user database
11077 entry associated with "jones". Similarly, the numeric value for the group ID is obtained by
11078 extracting the group ID from the group database entry associated with "cnd". This example
11079 assumes the calling program has appropriate privileges.

11080
```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <pwd.h>
#include <grp.h>
```

11085
```
struct passwd *pwd;
struct group  *grp;
int            fildes;
...
fildes = open("/home/cnd/mod1", O_RDWR);
```

```
pwd = getpwnam("jones");
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 1003.1-200x, **<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 defined as mandatory.

Issue 7

Functionality relating to XSI STREAMS is marked obsolescent.

11111 **NAME**

11112 fchownat — change owner and group of a file relative to directory file descriptor

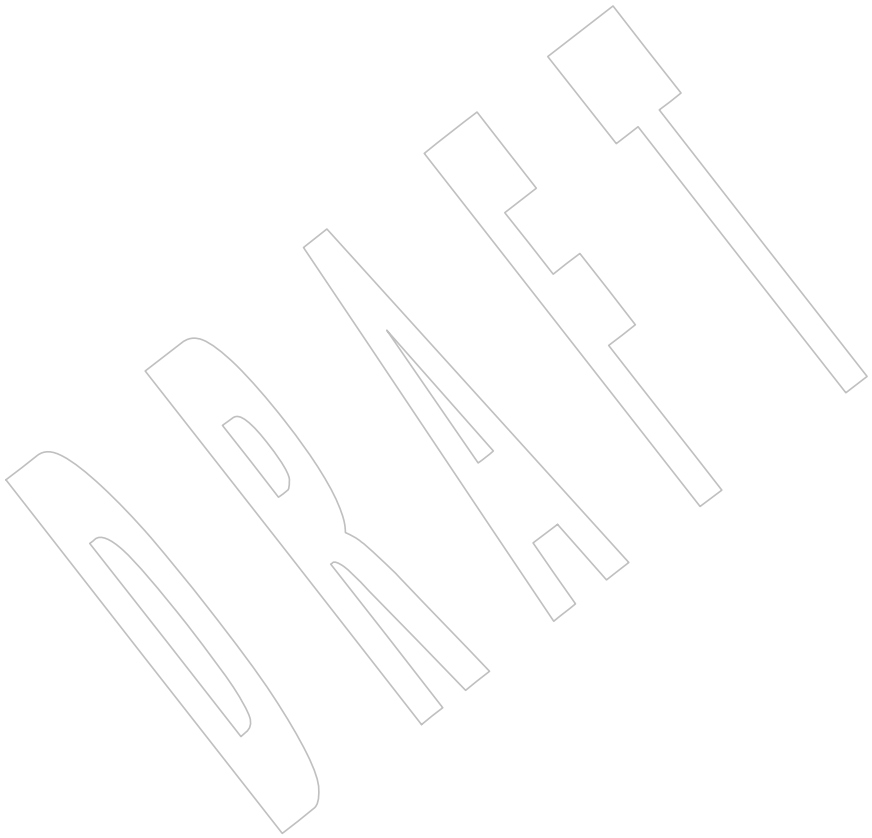
11113 **SYNOPSIS**

11114
```
#include <unistd.h>
```

11115
```
int fchownat(int fd, const char *path, uid_t owner, gid_t group,
    int flag);
```

11117 **DESCRIPTION**

11118 Refer to *chown( )*.



11119 **NAME**

11120 fclose — close a stream

11121 **SYNOPSIS**

11122 ```
#include <stdio.h>
```

11123 ```
int fclose(FILE *stream);
```

11124 **DESCRIPTION**

11125 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11126 conflict between the requirements described here and the ISO C standard is unintentional. This
11127 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

11128 The *fclose*( ) function shall cause the stream pointed to by *stream* to be flushed and the associated
11129 file to be closed. Any unwritten buffered data for the stream shall be written to the file; any
11130 unread buffered data shall be discarded. Whether or not the call succeeds, the stream shall be
11131 disassociated from the file and any buffer set by the *setbuf*( ) or *setvbuf*( ) function shall be
11132 disassociated from the stream. If the associated buffer was automatically allocated, it shall be
11133 deallocated.

11134 CX If the file is not already at EOF, and the file is one capable of seeking, the file offset of the
11135 underlying open file description shall be adjusted so that the next operation on the open file
11136 description deals with the byte after the last one read from or written to the stream being closed.

11137 The *fclose*( ) function shall mark for update the *st_ctime* and *st_mtime* fields of the underlying file,
11138 if the stream was writable, and if buffered data remains that has not yet been written to the file.
11139 The *fclose*( ) function shall perform the equivalent of a *close*( ) on the file descriptor that is
11140 associated with the stream pointed to by *stream*.

11141 After the call to *fclose*( ), any use of *stream* results in undefined behavior.

11142 **RETURN VALUE**

11143 CX Upon successful completion, *fclose*( ) shall return 0; otherwise, it shall return EOF and set *errno*
11144 to indicate the error.

11145 **ERRORS**

11146 The *fclose*( ) function shall fail if:

11147 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and
11148 the thread would be delayed in the write operation.

11149 CX [EBADF] The file descriptor underlying stream is not valid.

11150 CX [EFBIG] An attempt was made to write a file that exceeds the maximum file size.

11151 XSI [EFBIG] An attempt was made to write a file that exceeds the file size limit of the
11152 process.

11153 CX [EFBIG] The file is a regular file and an attempt was made to write at or beyond the
11154 offset maximum associated with the corresponding stream.

11155 CX [EINTR] The *fclose*( ) function was interrupted by a signal.

11156 CX [EIO] The process is a member of a background process group attempting to write to
11157 its controlling terminal, TOSTOP is set, the process is neither ignoring nor
11158 blocking SIGTTOU, and the process group of the process is orphaned. This
11159 error may also be returned under implementation-defined conditions.

CX [ENOMEM] The underlying stream was created by *open_memstream*( ) or *open_wmemstream*( ) and insufficient memory is available.

CX [ENOSPC] There was no free space remaining on the device containing the file or in the buffer used by the *fmemopen*( ) function.

CX [EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A SIGPIPE signal shall also be sent to the thread.

The *fclose*( ) function may fail if:

CX [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( ), *fmemopen*( ), *fopen*( ), *getrlimit*( ), *open_memstream*( ), *ulimit*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/28 is applied, updating the [EAGAIN] error in the ERRORS section from "the process would be delayed" to "the thread would be delayed".

Issue 7

Austin Group Interpretation 1003.1-2001 #002 is applied, clarifying the interaction of file descriptors and streams.

The [ENOSPC] error condition is updated and the [ENOMEM] error is added from The Open Group Technical Standard, 2006, Extended API Set Part 1.

11200 **NAME**

11201 fcntl — file control

11202 **SYNOPSIS**

11203
```
#include <fcntl.h>
```

11204
```
int fcntl(int fildes, int cmd, ...);
```

11205 **DESCRIPTION**

11206 The *fcntl*( ) function shall perform the operations described below on open files. The *fildes*
11207 argument is a file descriptor.

11208 The available values for *cmd* are defined in **<fcntl.h>** and are as follows:

11209 F_DUPFD Return a new file descriptor which shall be the lowest numbered available
11210 (that is, not already open) file descriptor greater than or equal to the third
11211 argument, *arg*, taken as an integer of type **int**. The new file descriptor shall
11212 refer to the same open file description as the original file descriptor, and shall
11213 share any locks. The FD_CLOEXEC flag associated with the new file
11214 descriptor shall be cleared to keep the file open across calls to one of the *exec*
11215 functions.

11216 F_GETFD Get the file descriptor flags defined in **<fcntl.h>** that are associated with the
11217 file descriptor *fildes*. File descriptor flags are associated with a single file
11218 descriptor and do not affect other file descriptors that refer to the same file.

11219 F_SETFD Set the file descriptor flags defined in **<fcntl.h>**, that are associated with *fildes*,
11220 to the third argument, *arg*, taken as type **int**. If the FD_CLOEXEC flag in the
11221 third argument is 0, the file shall remain open across the *exec* functions;
11222 otherwise, the file shall be closed upon successful execution of one of the *exec*
11223 functions.

11224 F_GETFL Get the file status flags and file access modes, defined in **<fcntl.h>**, for the file
11225 description associated with *fildes*. The file access modes can be extracted from
11226 the return value using the mask O_ACCMODE, which is defined in **<fcntl.h>**.
11227 File status flags and file access modes are associated with the file description
11228 and do not affect other file descriptors that refer to the same file with different
11229 open file descriptions.

11230 F_SETFL Set the file status flags, defined in **<fcntl.h>**, for the file description associated
11231 with *fildes* from the corresponding bits in the third argument, *arg*, taken as
11232 type **int**. Bits corresponding to the file access mode and the file creation flags,
11233 as defined in **<fcntl.h>**, that are set in *arg* shall be ignored. If any bits in *arg*
11234 other than those mentioned here are changed by the application, the result is
11235 unspecified.

11236 F_GETOWN If *fildes* refers to a socket, get the process or process group ID specified to
11237 receive SIGURG signals when out-of-band data is available. Positive values
11238 indicate a process ID; negative values, other than –1, indicate a process group
11239 ID. If *fildes* does not refer to a socket, the results are unspecified.

11240 F_SETOWN If *fildes* refers to a socket, set the process or process group ID specified to
11241 receive SIGURG signals when out-of-band data is available, using the value of
11242 the third argument, *arg*, taken as type **int**. Positive values indicate a process
11243 ID; negative values, other than –1, indicate a process group ID. If *fildes* does
11244 not refer to a socket, the results are unspecified.

11245 The following values for *cmd* are available for advisory record locking. Record locking shall be

supported for regular files, and may be supported for other files.

F_GETLK Get the first lock which blocks the lock description pointed to by the third argument, *arg*, taken as a pointer to type **struct flock**, defined in **<fcntl.h>**. The information retrieved shall overwrite the information passed to *fcntl*( ) in the structure **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.

F_SETLK Set or clear a file segment lock according to the lock description pointed to by the third argument, *arg*, taken as a pointer to type **struct flock**, defined in **<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 **<fcntl.h>**. If a shared or exclusive lock cannot be set, *fcntl*( ) shall return immediately with a return value of –1.

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 *fcntl*( ) is waiting for a region, *fcntl*( ) shall be interrupted. Upon return from the signal handler, *fcntl*( ) shall return –1 with *errno* set to [EINTR], and the lock operation shall not be done.

Additional implementation-defined values for *cmd* may be defined in **<fcntl.h>**. Their names shall start with 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 **flock** describes the type (*l_type*), starting offset (*l_whence*), relative offset (*l_start*), size (*l_len*), and process ID (*l_pid*) of the segment of the file to be affected.

The value of *l_whence* is SEEK_SET, SEEK_CUR, or SEEK_END, to indicate that the relative offset *l_start* bytes shall be measured from the start of the file, current position, or end of the file, respectively. The value of *l_len* is the number of consecutive bytes to be locked. The value of *l_len* may be negative (where the definition of **off_t** permits negative values of *l_len*). The *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 **flock** structure shall be as follows:

l_type Type of blocking lock found.

l_whence SEEK_SET.

l_start Start of the blocking lock.

l_len Length of the blocking lock.

l_pid Process ID of the process that holds the blocking lock.

If the command is F_SETLKW and the process must wait for another process to release a lock, then the range of bytes to be locked shall be determined before the *fcntl*( ) function blocks. If the file size or file descriptor seek offset change while *fcntl*( ) is blocked, this shall not affect the range of bytes locked.

If *l_len* is positive, the area affected shall start at *l_start* and end at *l_start*+*l_len*–1. If *l_len* is negative, the area affected shall start at *l_start*+*l_len* and end at *l_start*–1. Locks may start and extend beyond the current end of a file, but shall not extend before the beginning of the file. A lock shall be set to extend to the largest possible value of the file offset for that file by setting *l_len* to 0. If such a lock also has *l_start* set to 0 and *l_whence* is set to SEEK_SET, the whole file shall be locked.

There shall be at most one type of lock set for each byte in the file. Before a successful return from an F_SETLK or an F_SETLKW request when the calling process has previously existing locks on bytes in the region specified by the request, the previous lock type for each byte in the specified region shall be replaced by the new lock type. As specified above under the descriptions of shared locks and exclusive locks, an F_SETLK or an F_SETLKW request (respectively) shall fail or block when another process has existing locks on bytes in the specified region and the type of any of those locks conflicts with the type specified in the request.

All locks associated with a file for a given process shall be removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process.

A potential for deadlock occurs if a process controlling a locked region is put to sleep by attempting to lock the locked region of another process. If the system detects that sleeping until a locked region is unlocked would cause a deadlock, *fcntl*( ) shall fail with an [EDEADLK] error.

An unlock (F_UNLCK) request in which *l_len* is non-zero and the offset of the last byte of the requested segment is the maximum value for an object of type **off_t**, when the process has an existing lock in which *l_len* is 0 and which includes the last byte of the requested segment, shall be treated as a request to unlock from the start of the requested segment with an *l_len* equal to 0. Otherwise, an unlock (F_UNLCK) request shall attempt to unlock only the requested segment.

SHM When the file descriptor *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.

TYM 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.
F_SETOWN	Value other than –1.

Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *fcntl*( ) function shall fail if:

[EACCES] or [EAGAIN]
: 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 argument *cmd* is F_DUPFD and all file descriptors available to the process are currently open, 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.

EXAMPLES

Locking and Unlocking a File

The following example demonstrates how to place a lock on bytes 100 to 109 of a file and then later remove it. F_SETLK is used to perform a non-blocking lock request so that the process does not have to wait if an incompatible lock is held by another process; instead the process can take some other action.

```
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>

int
main(int argc, char *argv[])
{
    int fd;
```

```
        struct flock fl;

        fd = open("testfile", O_RDWR);
        if (fd == -1)
            /* Handle error */;

        /* Make a non-blocking request to place a write lock
           on bytes 100-109 of testfile */

        fl.l_type = F_WRLCK;
        fl.l_whence = SEEK_SET;
        fl.l_start = 100;
        fl.l_len = 10;

        if (fcntl(fd, F_SETLK, &fl) == –1) {
            if (errno == EACCES || errno == EAGAIN) {
                printf("Already locked by another process\n");

                /* We can't get the lock at the moment */

            } else {
                /* Handle unexpected error */;
            }
        } else { /* Lock was granted... */

            /* Perform I/O on bytes 100 to 109 of file */

            /* Unlock the locked bytes */

            fl.l_type = F_UNLCK;
            fl.l_whence = SEEK_SET;
            fl.l_start = 100;
            fl.l_len = 10;
            if (fcntl(fd, F_SETLK, &fl) == –1)
                /* Handle error */;
        }
        exit(EXIT_SUCCESS);
    } /* main */
```

Setting the Close-on-Exec Flag

The following example demonstrates how to set the close-on-exec flag for the file descriptor *fd*.

```
#include <unistd.h>
#include <fcntl.h>
...
    int flags;

    flags = fcntl(fd, F_GETFD);
    if (flags == –1)
        /* Handle error */;
    flags |= FD_CLOEXEC;
    if (fcntl(fd, F_SETFD, flags) == –1)
        /* Handle error */;"
```

APPLICATION USAGE

None.

RATIONALE

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 on them, rather than assuming that only the values defined by this volume of IEEE Std 1003.1-200x are valid. It is a common error to forget this, particularly in the case of F_SETFD.

This volume of IEEE Std 1003.1-200x permits concurrent read and write access to file data using the *fcntl*( ) function; this is a change from the 1984 /usr/group standard and early proposals. Without concurrency controls, this feature may not be fully utilized without occasional loss of data.

Data losses occur in several ways. One case occurs when several processes try to update the same record, without sequencing controls; several updates may occur in parallel and the last writer "wins". Another case is a bit-tree or other internal list-based database that is undergoing reorganization. Without exclusive use to the tree segment by the updating process, other reading processes chance getting lost in the database when the index blocks are split, condensed, inserted, or deleted. While *fcntl*( ) is useful for many applications, it is not intended to be overly general and does not handle the bit-tree example well.

This facility is only required for regular files because it is not appropriate for many devices such as terminals and network connections.

Since *fcntl*( ) works with "any file descriptor associated with that file, however it is obtained", the file descriptor may have been inherited through a *fork*( ) or *exec* operation and thus may affect a file that another process also has open.

The use of the open file description to identify what to lock requires extra calls and presents problems if several processes are sharing an open file description, but there are too many implementations of the existing mechanism for this volume of IEEE Std 1003.1-200x to use different specifications.

Another consequence of this model is that closing any file descriptor for a given file (whether or not it is the same open file description that created the lock) causes the locks on that file to be relinquished for that process. Equivalently, any close for any file/process pair relinquishes the locks owned on that file for that process. But note that while an open file description may be shared through *fork*( ), locks are not inherited through *fork*( ). Yet locks may be inherited through one of the *exec* functions.

The identification of a machine in a network environment is outside the scope of this volume of IEEE Std 1003.1-200x. Thus, an *l_sysid* member, such as found in System V, is not included in the locking structure.

Changing of lock types can result in a previously locked region being split into smaller regions.

Mandatory locking was a major feature of the 1984 /usr/group standard.

For advisory file record locking to be effective, all processes that have access to a file must cooperate and use the advisory mechanism before doing I/O on the file. Enforcement-mode record locking is important when it cannot be assumed that all processes are cooperating. For example, if one user uses an editor to update a file at the same time that a second user executes another process that updates the same file and if only one of the two processes is using advisory locking, the processes are not cooperating. Enforcement-mode record locking would protect against accidental collisions.

Secondly, advisory record locking requires a process using locking to bracket each I/O operation

with lock (or test) and unlock operations. With enforcement-mode file and record locking, a process can lock the file once and unlock when all I/O operations have been completed. Enforcement-mode record locking provides a base that can be enhanced; for example, with sharable locks. That is, the mechanism could be enhanced to allow a process to lock a file so other processes could read it, but none of them could write it.

Mandatory locks were omitted for several reasons:

1. Mandatory lock setting was done by multiplexing the set-group-ID bit in most implementations; this was confusing, at best.
2. The relationship to file truncation as supported in 4.2 BSD was not well specified.
3. Any publicly readable file could be locked by anyone. Many historical implementations keep the password database in a publicly readable file. A malicious user could thus prohibit logins. Another possibility would be to hold open a long-distance telephone line.
4. Some demand-paged historical implementations offer memory mapped files, and enforcement cannot be done on that type of file.

Since sleeping on a region is interrupted with any signal, *alarm*( ) may be used to provide a timeout facility in applications requiring it. This is useful in deadlock detection. Since implementation of full deadlock detection is not always feasible, the [EDEADLK] error was made optional.

FUTURE DIRECTIONS

None.

SEE ALSO

alarm( ), *close*( ), *exec*, *open*( ), *sigaction*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fcntl.h>**, **<signal.h>**

CHANGE HISTORY

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

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Large File Summit extensions are added.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

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

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- In the DESCRIPTION, sentences describing behavior when *l_len* is negative are now mandated, and the description of unlock (F_UNLOCK) when *l_len* is non-negative is mandated.
- In the ERRORS section, the [EINVAL] error condition has the case mandated when the *cmd* is invalid, and two [EOVERFLOW] error conditions are added.

The F_GETOWN and F_SETOWN values are added for sockets.

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

- Clarification is added that the extent of the bytes locked is determined prior to the blocking action.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that *fcntl*( ) results are unspecified for typed memory objects.

The normative text is updated to avoid use of the term "must" for application requirements.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/29 is applied, adding the example to the EXAMPLES section.

Issue 7

The optional **<unistd.h>** header is removed from this function, since **<fcntl.h>** now defines SEEK_SET, SEEK_CUR, and SEEK_END as part of the Base.

11524 **NAME**

11525 fdatasync — synchronize the data of a file (**REALTIME**)

11526 **SYNOPSIS**

11527 SIO
```
#include <unistd.h>

int fdatasync(int fildes);
```

11528

11529 **DESCRIPTION**

11530 The *fdatasync*( ) function shall force all currently queued I/O operations associated with the file
11531 indicated by file descriptor *fildes* to the synchronized I/O completion state.

11532 The functionality shall be equivalent to *fsync*( ) with the symbol _POSIX_SYNCHRONIZED_IO
11533 defined, with the exception that all I/O operations shall be completed as defined for
11534 synchronized I/O data integrity completion.

11535 **RETURN VALUE**

11536 If successful, the *fdatasync*( ) function shall return the value 0; otherwise, the function shall return
11537 the value –1 and set *errno* to indicate the error. If the *fdatasync*( ) function fails, outstanding I/O
11538 operations are not guaranteed to have been completed.

11539 **ERRORS**

11540 The *fdatasync*( ) function shall fail if:

11541 [EBADF] The *fildes* argument is not a valid file descriptor open for writing.

11542 [EINVAL] This implementation does not support synchronized I/O for this file.

11543 In the event that any of the queued I/O operations fail, *fdatasync*( ) shall return the error
11544 conditions defined for *read*( ) and *write*( ).

11545 **EXAMPLES**

11546 None.

11547 **APPLICATION USAGE**

11548 None.

11549 **RATIONALE**

11550 None.

11551 **FUTURE DIRECTIONS**

11552 None.

11553 **SEE ALSO**

11554 *aio_fsync*( ), *fcntl*( ), *fsync*( ), *open*( ), *read*( ), *write*( ), the Base Definitions volume of
11555 IEEE Std 1003.1-200x, **<unistd.h>**

11556 **CHANGE HISTORY**

11557 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

11558 **Issue 6**

11559 The [ENOSYS] error condition has been removed as stubs need not be provided if an
11560 implementation does not support the Synchronized Input and Output option.

11561 The *fdatasync*( ) function is marked as part of the Synchronized Input and Output option.

11562 **NAME**

11563 fdetach — detach a name from a STREAMS-based file descriptor (**STREAMS**)

11564 **SYNOPSIS**

11565 OB XSR
```
#include <stropts.h>

int fdetach(const char *path);
```

11567 **DESCRIPTION**

11568 The *fdetach*( ) function shall detach a STREAMS-based file from the file to which it was attached
11569 by a previous call to *fattach*( ). The *path* argument points to the pathname of the attached
11570 STREAMS file. The process shall have appropriate privileges or be the owner of the file. A
11571 successful call to *fdetach*( ) shall cause all pathnames that named the attached STREAMS file to
11572 again name the file to which the STREAMS file was attached. All subsequent operations on *path*
11573 shall operate on the underlying file and not on the STREAMS file.

11574 All open file descriptions established while the STREAMS file was attached to the file referenced
11575 by *path* shall still refer to the STREAMS file after the *fdetach*( ) has taken effect.

11576 If there are no open file descriptors or other references to the STREAMS file, then a successful
11577 call to *fdetach*( ) shall be equivalent to performing the last *close*( ) on the attached file.

11578 **RETURN VALUE**

11579 Upon successful completion, *fdetach*( ) shall return 0; otherwise, it shall return –1 and set *errno* to
11580 indicate the error.

11581 **ERRORS**

11582 The *fdetach*( ) function shall fail if:

11583 [EACCES] Search permission is denied on a component of the path prefix.

11584 [EINVAL] The *path* argument names a file that is not currently attached.

11585 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
11586 argument.

11587 [ENAMETOOLONG]
11588 The size of a pathname exceeds {PATH_MAX} or a pathname component is
11589 longer than {NAME_MAX}.

11590 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

11591 [ENOTDIR] A component of the path prefix is not a directory.

11592 [EPERM] The effective user ID is not the owner of *path* and the process does not have
11593 appropriate privileges.

11594 The *fdetach*( ) function may fail if:

11595 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
11596 resolution of the *path* argument.

11597 [ENAMETOOLONG]
11598 Pathname resolution of a symbolic link produced an intermediate result
11599 whose length exceeds {PATH_MAX}.

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.

```
#include <stropts.h>
...
    char *filename = "/tmp/named-STREAM";
    int ret;

    ret = fdetach(filename);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

The *fdetach*( ) function may be removed in a future version.

SEE ALSO

fattach( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stropts.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The normative text 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.

Issue 7

The *fdetach*( ) function is marked obsolescent.

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

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 1003.1-200x 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*, +0 is returned.

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 positive difference value.

If *x*–*y* is positive and overflows, a range error shall occur and *fdim*( ), *fdimf*( ), and *fdiml*( ) shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

XSI 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.

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

ERRORS

The *fdim*( ) function 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.

The *fdim*( ) 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

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 option 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 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

11682 **NAME**

11683 fdopen — associate a stream with a file descriptor

11684 **SYNOPSIS**

11685 CX
```
#include <stdio.h>

FILE *fdopen(int fildes, const char *mode);
```

11687 **DESCRIPTION**

11688 The *fdopen*( ) function shall associate a stream with a file descriptor.

11689 The *mode* argument is a character string having one of the following values:

11690 *r* or *rb* Open a file for reading.

11691 *w* or *wb* Open a file for writing.

11692 *a* or *ab* Open a file for writing at end-of-file.

11693 *r+* or *rb+* or *r+b* Open a file for update (reading and writing).

11694 *w+* or *wb+* or *w+b* Open a file for update (reading and writing).

11695 *a+* or *ab+* or *a+b* Open a file for update (reading and writing) at end-of-file.

11696 The meaning of these flags is exactly as specified in *fopen*( ), except that modes beginning with *w*
11697 shall not cause truncation of the file.

11698 Additional values for the *mode* argument may be supported by an implementation.

11699 The application shall ensure that the mode of the stream as expressed by the *mode* argument is
11700 allowed by the file access mode of the open file description to which *fildes* refers. The file
11701 position indicator associated with the new stream is set to the position indicated by the file offset
11702 associated with the file descriptor.

11703 The error and end-of-file indicators for the stream shall be cleared. The *fdopen*( ) function may
11704 cause the *st_atime* field of the underlying file to be marked for update.

11705 SHM If *fildes* refers to a shared memory object, the result of the *fdopen*( ) function is unspecified.

11706 TYM If *fildes* refers to a typed memory object, the result of the *fdopen*( ) function is unspecified.

11707 The *fdopen*( ) function shall preserve the offset maximum previously set for the open file
11708 description corresponding to *fildes*.

11709 **RETURN VALUE**

11710 Upon successful completion, *fdopen*( ) shall return a pointer to a stream; otherwise, a null pointer
11711 shall be returned and *errno* set to indicate the error.

11712 **ERRORS**

11713 The *fdopen*( ) function may fail if:

11714 [EBADF] The *fildes* argument is not a valid file descriptor.

11715 [EINVAL] The *mode* argument is not a valid mode.

11716 [EMFILE] {FOPEN_MAX} streams are currently open in the calling process.

11717 [EMFILE] {STREAM_MAX} streams are currently open in the calling process.

11718 [ENOMEM] Insufficient space to allocate a buffer.

EXAMPLES

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*( ), *fcntl*( ), or *socket*( ); inherited through *fork*( ), *posix_spawn*( ), or *exec*; or perhaps obtained by other means.

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 1003.1-200x, 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*( ), *fmemopen*( ), *fopen*( ), *open*( ), *open_memstream*( ), *posix_spawn*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/30 is applied, making corrections to the RATIONALE.

NAME

fdopendir, opendir — open directory associated with file descriptor

SYNOPSIS

```
#include <dirent.h>

DIR *fdopendir(int fd);
DIR *opendir(const char *dirname);
```

DESCRIPTION

The *fdopendir*( ) function shall be equivalent to the *opendir*( ) function except that the directory is specified by a file descriptor rather than by a name. The file offset associated with the file descriptor at the time of the call determines which entries are returned.

Upon successful return from *fdopendir*( ), the file descriptor is under the control of the system, and if any attempt is made to close the file descriptor, or to modify the state of the associated description other than by means of *closedir*( ), *readdir*( ), *readdir_r*( ), or *rewinddir*( ), the behavior is implementation-defined. Upon calling *closedir*( ) the file descriptor shall be closed.

It is unspecified whether the FD_CLOEXEC flag will be set on the file descriptor by a successful call to *fdopendir*( ).

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.

If the type **DIR** is implemented using a file descriptor, the descriptor shall be obtained as if the O_DIRECTORY flag was passed to *open*( ).

RETURN VALUE

Upon successful completion, these functions shall return a pointer to an object of type **DIR**. Otherwise, these functions shall return a null pointer and set *errno* to indicate the error.

ERRORS

The *fdopendir*( ) function shall fail if:

[EBADF] The *fd* argument is not a valid file descriptor open for searching.

[ENOTDIR] The descriptor *fd* is not associated with a directory.

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] All file descriptors available to the process are currently open.

[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.

```
#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

The purpose of the *fdopendir*( ) function is to enable opening files in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to *opendir*( ), resulting in unspecified behavior.

Based on historical implementations, the rules about file descriptors apply to directory streams as well. However, this volume of IEEE Std 1003.1-200x 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 1003.1-200x 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 1003.1-200x 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( ), *dirfd*( ), *fstatat*( ), *open*( ), *readdir*( ), *rewinddir*( ), *symlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<dirent.h>**, **<limits.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.
- 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.

Issue 7

SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

The *fdopendir*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 2.

11874 **NAME**

11875 feclearexcept — clear floating-point exception

11876 **SYNOPSIS**

11877
```
#include <fenv.h>
```

11878
```
int feclearexcept(int excepts);
```

11879 **DESCRIPTION**

11880 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11881 conflict between the requirements described here and the ISO C standard is unintentional. This
11882 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

11883 The *feclearexcept*( ) function shall attempt to clear the supported floating-point exceptions
11884 represented by *excepts*.

11885 **RETURN VALUE**

11886 If the argument is zero or if all the specified exceptions were successfully cleared, *feclearexcept*( )
11887 shall return zero. Otherwise, it shall return a non-zero value.

11888 **ERRORS**

11889 No errors are defined.

11890 **EXAMPLES**

11891 None.

11892 **APPLICATION USAGE**

11893 None.

11894 **RATIONALE**

11895 None.

11896 **FUTURE DIRECTIONS**

11897 None.

11898 **SEE ALSO**

11899 *fegetexceptflag*( ), *feraiseexcept*( ), *fesetexceptflag*( ), *fetestexcept*( ), the Base Definitions volume of
11900 IEEE Std 1003.1-200x, **<fenv.h>**

11901 **CHANGE HISTORY**

11902 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

11903 ISO/IEC 9899: 1999 standard, Technical Corrigendum 1 is incorporated.

11904 **NAME**

11905 fegetenv, fesetenv — get and set current floating-point environment

11906 **SYNOPSIS**

11907
```
#include <fenv.h>
```

11908
```
int fegetenv(fenv_t *envp);
```
11909
```
int fesetenv(const fenv_t *envp);
```

11910 **DESCRIPTION**

11911 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11912 conflict between the requirements described here and the ISO C standard is unintentional. This
11913 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

11914 The *fegetenv*( ) function shall attempt to store the current floating-point environment in the object
11915 pointed to by *envp*.

11916 The *fesetenv*( ) function shall attempt to establish the floating-point environment represented by
11917 the object pointed to by *envp*. The argument *envp* shall point to an object set by a call to
11918 *fegetenv*( ) or *feholdexcept*( ), or equal a floating-point environment macro. The *fesetenv*( ) function
11919 does not raise floating-point exceptions, but only installs the state of the floating-point status
11920 flags represented through its argument.

11921 **RETURN VALUE**

11922 If the representation was successfully stored, *fegetenv*( ) shall return zero. Otherwise, it shall
11923 return a non-zero value. If the environment was successfully established, *fesetenv*( ) shall return
11924 zero. Otherwise, it shall return a non-zero value.

11925 **ERRORS**

11926 No errors are defined.

11927 **EXAMPLES**

11928 None.

11929 **APPLICATION USAGE**

11930 None.

11931 **RATIONALE**

11932 None.

11933 **FUTURE DIRECTIONS**

11934 None.

11935 **SEE ALSO**

11936 *feholdexcept*( ), *feupdateenv*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fenv.h>**

11937 **CHANGE HISTORY**

11938 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

11939 ISO/IEC 9899: 1999 standard, Technical Corrigendum 1 is incorporated.

NAME

fegetexceptflag, fesetexceptflag — get and set floating-point status flags

SYNOPSIS

```
#include <fenv.h>

int fegetexceptflag(fexcept_t *flagp, int excepts);
int fesetexceptflag(const fexcept_t *flagp, int excepts);
```

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 1003.1-200x defers to the ISO C standard.

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 1003.1-200x, **<fenv.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

ISO/IEC 9899: 1999 standard, Technical Corrigendum 1 is incorporated.

NAME

fegetround, fesetround — get and set current rounding direction

SYNOPSIS

```
#include <fenv.h>

int fegetround(void);
int fesetround(int round);
```

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 1003.1-200x defers to the ISO C standard.

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 <assert.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.

12022 **FUTURE DIRECTIONS**

12023 None.

12024 **SEE ALSO**

12025 The Base Definitions volume of IEEE Std 1003.1-200x, **<fenv.h>**

12026 **CHANGE HISTORY**

12027 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

12028 ISO/IEC 9899: 1999 standard, Technical Corrigendum 1 is incorporated.

12029 **NAME**

12030 feholdexcept — save current floating-point environment

12031 **SYNOPSIS**

12032
```
#include <fenv.h>
```

12033
```
int feholdexcept(fenv_t *envp);
```

12034 **DESCRIPTION**

12035 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12036 conflict between the requirements described here and the ISO C standard is unintentional. This
12037 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

12038 The *feholdexcept*( ) function shall save the current floating-point environment in the object
12039 pointed to by *envp*, clear the floating-point status flags, and then install a non-stop (continue on
12040 floating-point exceptions) mode, if available, for all floating-point exceptions.

12041 **RETURN VALUE**

12042 The *feholdexcept*( ) function shall return zero if and only if non-stop floating-point exception
12043 handling was successfully installed.

12044 **ERRORS**

12045 No errors are defined.

12046 **EXAMPLES**

12047 None.

12048 **APPLICATION USAGE**

12049 None.

12050 **RATIONALE**

12051 The *feholdexcept*( ) function should be effective on typical IEC 60559: 1989 standard
12052 implementations which have the default non-stop mode and at least one other mode for trap
12053 handling or aborting. If the implementation provides only the non-stop mode, then installing the
12054 non-stop mode is trivial.

12055 **FUTURE DIRECTIONS**

12056 None.

12057 **SEE ALSO**

12058 *fegetenv*( ), *fesetenv*( ), *feupdateenv*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
12059 **<fenv.h>**

12060 **CHANGE HISTORY**

12061 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

12062 **NAME**

12063 feof — test end-of-file indicator on a stream

12064 **SYNOPSIS**

12065
```
#include <stdio.h>
```

12066
```
int feof(FILE *stream);
```

12067 **DESCRIPTION**

12068 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12069 conflict between the requirements described here and the ISO C standard is unintentional. This
12070 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

12071 The *feof*( ) function shall test the end-of-file indicator for the stream pointed to by *stream*.

12072 **RETURN VALUE**

12073 The *feof*( ) function shall return non-zero if and only if the end-of-file indicator is set for *stream*.

12074 **ERRORS**

12075 No errors are defined.

12076 **EXAMPLES**

12077 None.

12078 **APPLICATION USAGE**

12079 None.

12080 **RATIONALE**

12081 None.

12082 **FUTURE DIRECTIONS**

12083 None.

12084 **SEE ALSO**

12085 *clearerr*( ), *ferror*( ), *fopen*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

12086 **CHANGE HISTORY**

12087 First released in Issue 1. Derived from Issue 1 of the SVID.

12088 **NAME**

12089 feraiseexcept — raise floating-point exception

12090 **SYNOPSIS**

12091
```
#include <fenv.h>
```

12092
```
int feraiseexcept(int excepts);
```

12093 **DESCRIPTION**

12094 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12095 conflict between the requirements described here and the ISO C standard is unintentional. This
12096 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

12097 The *feraiseexcept*( ) function shall attempt to raise the supported floating-point exceptions
12098 represented by the argument *excepts*. The order in which these floating-point exceptions are
12099 raised is unspecified. Whether the *feraiseexcept*( ) function additionally raises the inexact floating-
12100 point exception whenever it raises the overflow or underflow floating-point exception is
12101 implementation-defined.

12102 **RETURN VALUE**

12103 If the argument is zero or if all the specified exceptions were successfully raised, *feraiseexcept*( )
12104 shall return zero. Otherwise, it shall return a non-zero value.

12105 **ERRORS**

12106 No errors are defined.

12107 **EXAMPLES**

12108 None.

12109 **APPLICATION USAGE**

12110 The effect is intended to be similar to that of floating-point exceptions raised by arithmetic
12111 operations. Hence, enabled traps for floating-point exceptions raised by this function are taken.

12112 **RATIONALE**

12113 Raising overflow or underflow is allowed to also raise inexact because on some architectures the
12114 only practical way to raise an exception is to execute an instruction that has the exception as a
12115 side effect. The function is not restricted to accept only valid coincident expressions for atomic
12116 operations, so the function can be used to raise exceptions accrued over several operations.

12117 **FUTURE DIRECTIONS**

12118 None.

12119 **SEE ALSO**

12120 *feclearexcept*( ), *fegetexceptflag*( ), *fesetexceptflag*( ), *fetestexcept*( ), the Base Definitions volume of
12121 IEEE Std 1003.1-200x, **<fenv.h>**

12122 **CHANGE HISTORY**

12123 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

12124 ISO/IEC 9899: 1999 standard, Technical Corrigendum 1 is incorporated.

12125 **NAME**

12126 ferror — test error indicator on a stream

12127 **SYNOPSIS**

12128 ```
#include <stdio.h>
```

12129 ```
int ferror(FILE *stream);
```

12130 **DESCRIPTION**

12131 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12132 conflict between the requirements described here and the ISO C standard is unintentional. This
12133 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

12134 The *ferror*( ) function shall test the error indicator for the stream pointed to by *stream*.

12135 **RETURN VALUE**

12136 The *ferror*( ) function shall return non-zero if and only if the error indicator is set for *stream*.

12137 **ERRORS**

12138 No errors are defined.

12139 **EXAMPLES**

12140 None.

12141 **APPLICATION USAGE**

12142 None.

12143 **RATIONALE**

12144 None.

12145 **FUTURE DIRECTIONS**

12146 None.

12147 **SEE ALSO**

12148 *clearerr*( ), *feof*( ), *fopen*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

12149 **CHANGE HISTORY**

12150 First released in Issue 1. Derived from Issue 1 of the SVID.

fesetenv( )

12151 **NAME**

12152 fesetenv — set current floating-point environment

12153 **SYNOPSIS**

12154 ```
#include <fenv.h>
```

12155 ```
int fesetenv(const fenv_t *envp);
```

12156 **DESCRIPTION**

12157 Refer to *fegetenv( )*.

12158 **NAME**

12159 fesetexceptflag — set floating-point status flags

12160 **SYNOPSIS**

12161 `#include <fenv.h>`

12162 `int fesetexceptflag(const fexcept_t *flagp, int excepts);`

12163 **DESCRIPTION**

12164 Refer to *fegetexceptflag( )*.

fesetround( )

12165 **NAME**

12166 fesetround — set current rounding direction

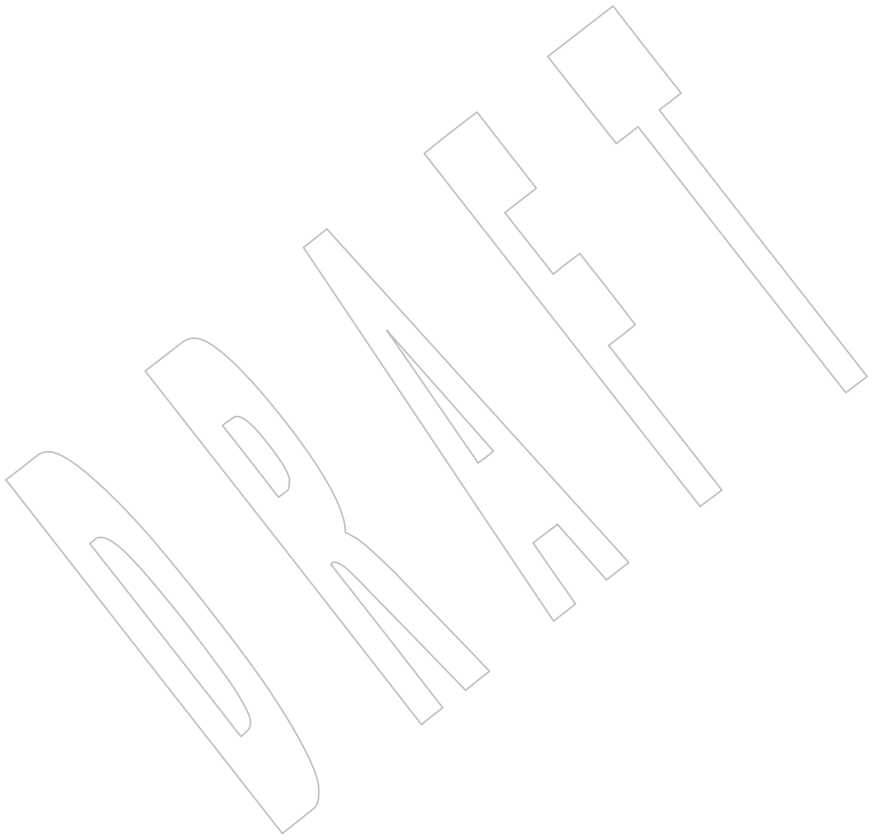
12167 **SYNOPSIS**

12168
```
#include <fenv.h>
```

12169
```
int fesetround(int round);
```

12170 **DESCRIPTION**

12171 Refer to *fegetround( )*.



12172 **NAME**

12173 fetestexcept — test floating-point exception flags

12174 **SYNOPSIS**

12175
```
#include <fenv.h>
```

12176
```
int fetestexcept(int excepts);
```

12177 **DESCRIPTION**

12178 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12179 conflict between the requirements described here and the ISO C standard is unintentional. This
12180 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

12181 The *fetestexcept*( ) function shall determine which of a specified subset of the floating-point
12182 exception flags are currently set. The *excepts* argument specifies the floating-point status flags to
12183 be queried.

12184 **RETURN VALUE**

12185 The *fetestexcept*( ) function shall return the value of the bitwise-inclusive OR of the floating-point
12186 exception macros corresponding to the currently set floating-point exceptions included in
12187 *excepts*.

12188 **ERRORS**

12189 No errors are defined.

12190 **EXAMPLES**

12191 The following example calls function *f*( ) if an invalid exception is set, and then function *g*( ) if an
12192 overflow exception is set:

12193
```
#include <fenv.h>
```
12194
```
/* ... */
```
12195
```
{
```
12196
```
    #pragma STDC FENV_ACCESS ON
```
12197
```
    int set_excepts;
```
12198
```
    feclearexcept(FE_INVALID | FE_OVERFLOW);
```
12199
```
    // maybe raise exceptions
```
12200
```
    set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
```
12201
```
    if (set_excepts & FE_INVALID) f();
```
12202
```
    if (set_excepts & FE_OVERFLOW) g();
```
12203
```
    /* ... */
```
12204
```
}
```

12205 **APPLICATION USAGE**

12206 None.

12207 **RATIONALE**

12208 None.

12209 **FUTURE DIRECTIONS**

12210 None.

12211 **SEE ALSO**

12212 *feclearexcept*( ), *fegetexceptflag*( ), *feraiseexcept*( ), the Base Definitions volume of
12213 IEEE Std 1003.1-200x, **<fenv.h>**

12214 **CHANGE HISTORY**

12215 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

12216 **NAME**

12217 feupdateenv — update floating-point environment

12218 **SYNOPSIS**

12219
```
#include <fenv.h>
```

12220
```
int feupdateenv(const fenv_t *envp);
```

12221 **DESCRIPTION**

12222 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12223 conflict between the requirements described here and the ISO C standard is unintentional. This
12224 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

12225 The *feupdateenv*( ) function shall attempt to save the currently raised floating-point exceptions in
12226 its automatic storage, attempt to install the floating-point environment represented by the object
12227 pointed to by *envp*, and then attempt to raise the saved floating-point exceptions. The argument
12228 *envp* shall point to an object set by a call to *feholdexcept*( ) or *fegetenv*( ), or equal a floating-point
12229 environment macro.

12230 **RETURN VALUE**

12231 The *feupdateenv*( ) function shall return a zero value if and only if all the required actions were
12232 successfully carried out.

12233 **ERRORS**

12234 No errors are defined.

12235 **EXAMPLES**

12236 The following example shows sample code to hide spurious underflow floating-point
12237 exceptions:

12238
```
#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;
}
```

12251 **APPLICATION USAGE**

12252 None.

12253 **RATIONALE**

12254 None.

12255 **FUTURE DIRECTIONS**

12256 None.

12257 **SEE ALSO**

12258 *fegetenv*( ), *feholdexcept*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fenv.h>**

12259 **CHANGE HISTORY**

12260 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

12261 ISO/IEC 9899: 1999 standard, Technical Corrigendum 1 is incorporated.

12262 **NAME**

12263 fexecve — execute a file

12264 **SYNOPSIS**

12265
```
#include <unistd.h>
```

12266
```
int fexecve(int fd, char *const argv[], char *const envp[]);
```

12267 **DESCRIPTION**

12268 Refer to *exec*.

12269 **NAME**

12270 fflush — flush a stream

12271 **SYNOPSIS**

12272 ```
#include <stdio.h>
```

12273 ```
int fflush(FILE *stream);
```

12274 **DESCRIPTION**

12275 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12276 conflict between the requirements described here and the ISO C standard is unintentional. This
12277 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

12278 If *stream* points to an output stream or an update stream in which the most recent operation was
12279 CX not input, *fflush*( ) shall cause any unwritten data for that stream to be written to the file, and the
12280 *st_ctime* and *st_mtime* fields of the underlying file shall be marked for update.

12281 If *stream* is a null pointer, *fflush*( ) shall perform this flushing action on all streams for which the
12282 behavior is defined above.

12283 CX For a stream open for reading, if the file is not already at EOF, and the file is one capable of
12284 seeking, the file offset of the underlying open file description shall be adjusted so that the next
12285 operation on the open file description deals with the byte after the last one read from or written
12286 to the stream being flushed.

12287 **RETURN VALUE**

12288 Upon successful completion, *fflush*( ) shall return 0; otherwise, it shall set the error indicator for
12289 CX the stream, return EOF, and set *errno* to indicate the error.

12290 **ERRORS**

12291 The *fflush*( ) function shall fail if:

12292 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and
12293 the thread would be delayed in the write operation.

12294 CX [EBADF] The file descriptor underlying *stream* is not valid.

12295 CX [EFBIG] An attempt was made to write a file that exceeds the maximum file size.

12296 XSI [EFBIG] An attempt was made to write a file that exceeds the file size limit of the
12297 process.

12298 CX [EFBIG] The file is a regular file and an attempt was made to write at or beyond the
12299 offset maximum associated with the corresponding stream.

12300 CX [EINTR] The *fflush*( ) function was interrupted by a signal.

12301 CX [EIO] The process is a member of a background process group attempting to write to
12302 its controlling terminal, TOSTOP is set, the process is neither ignoring nor
12303 blocking SIGTTOU, and the process group of the process is orphaned. This
12304 error may also be returned under implementation-defined conditions.

12305 CX [ENOMEM] The underlying stream was created by *open_memstream*( ) or
12306 *open_wmemstream*( ) and insufficient memory is available.

12307 CX [ENOSPC] There was no free space remaining on the device containing the file or in the
12308 buffer used by the *fmemopen*( ) function.

12309 CX [EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by
12310 any process. A SIGPIPE signal shall also be sent to the thread.

12311 The *fflush*( ) function may fail if:

12312 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the
12313 capabilities of the device.

12314 **EXAMPLES**

12315 **Sending Prompts to Standard Output**

12316 The following example uses *printf*( ) calls to print a series of prompts for information the user
12317 must enter from standard input. The *fflush*( ) calls force the output to standard output. The
12318 *fflush*( ) function is used because standard output is usually buffered and the prompt may not
12319 immediately be printed on the output or terminal. The *gets*( ) calls read strings from standard
12320 input and place the results in variables, for use later in the program.

```
12321 #include <stdio.h>
12322 ...
12323 char user[100];
12324 char oldpasswd[100];
12325 char newpasswd[100];
12326 ...
12327 printf("User name: ");
12328 fflush(stdout);
12329 gets(user);
12330 printf("Old password: ");
12331 fflush(stdout);
12332 gets(oldpasswd);
12333 printf("New password: ");
12334 fflush(stdout);
12335 gets(newpasswd);
12336 ...
```

12337 **APPLICATION USAGE**

12338 None.

12339 **RATIONALE**

12340 Data buffered by the system may make determining the validity of the position of the current
12341 file descriptor impractical. Thus, enforcing the repositioning of the file descriptor after *fflush*( )
12342 on streams open for *read*( ) is not mandated by IEEE Std 1003.1-200x.

12343 **FUTURE DIRECTIONS**

12344 None.

12345 **SEE ALSO**

12346 *fmemopen*( ), *getrlimit*( ), *open_memstream*( ), *ulimit*( ), the Base Definitions volume of
12347 IEEE Std 1003.1-200x, **<stdio.h>**

12348 **CHANGE HISTORY**

12349 First released in Issue 1. Derived from Issue 1 of the SVID.

12350 **Issue 5**

12351 Large File Summit extensions are added.

12352 **Issue 6**

12353 Extensions beyond the ISO C standard are marked.

12354 The following new requirements on POSIX implementations derive from alignment with the
12355 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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/31 is applied, updating the [EAGAIN] error in the ERRORS section from "the process would be delayed" to "the thread would be delayed".

Issue 7

Austin Group Interpretation 1003.1-2001 #002 is applied, clarifying the interaction of file descriptors and streams.

The [ENOSPC] error condition is updated and the [ENOMEM] error is added from The Open Group Technical Standard, 2006, Extended API Set Part 1.

12368 **NAME**

12369 ffs — find first set bit

12370 **SYNOPSIS**

12371 XSI
```
#include <strings.h>
```
12372
```
int ffs(int i);
```

12373 **DESCRIPTION**

12374 The *ffs*( ) function shall find the first bit set (beginning with the least significant bit) in *i*, and
12375 return the index of that bit. Bits are numbered starting at one (the least significant bit).

12376 **RETURN VALUE**

12377 The *ffs*( ) function shall return the index of the first bit set. If *i* is 0, then *ffs*( ) shall return 0.

12378 **ERRORS**

12379 No errors are defined.

12380 **EXAMPLES**

12381 None.

12382 **APPLICATION USAGE**

12383 None.

12384 **RATIONALE**

12385 None.

12386 **FUTURE DIRECTIONS**

12387 None.

12388 **SEE ALSO**

12389 The Base Definitions volume of IEEE Std 1003.1-200x, **<strings.h>**

12390 **CHANGE HISTORY**

12391 First released in Issue 4, Version 2.

12392 **Issue 5**

12393 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

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 1003.1-200x defers to the ISO C standard.

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*( ).

CX 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*( ), *fread*( ), *fscanf*( ), *getc*( ), *getchar*( ), *gets*( ), or *scanf*( ) using *stream* that returns data not supplied by a prior call to *ungetc*( ).

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 end-of-file indicator for the stream shall be set and *fgetc*( ) shall return EOF. If a read error occurs, CX 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:

CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the thread would be delayed in the *fgetc*( ) operation.

CX [EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for reading.

CX [EINTR] The read operation was terminated due to the receipt of a signal, and no data was transferred.

CX [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.

CX [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:

CX [ENOMEM] Insufficient storage space is available.

CX [ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

12436 **EXAMPLES**
12437 None.

12438 **APPLICATION USAGE**
12439 If the integer value returned by *fgetc*( ) is stored into a variable of type **char** and then compared
12440 against the integer constant EOF, the comparison may never succeed, because sign-extension of
12441 a variable of type **char** on widening to integer is implementation-defined.

12442 The *ferror*( ) or *feof*( ) functions must be used to distinguish between an error condition and an
12443 end-of-file condition.

12444 **RATIONALE**
12445 None.

12446 **FUTURE DIRECTIONS**
12447 None.

12448 **SEE ALSO**
12449 *feof*( ), *ferror*( ), *fgets*( ), *fread*( ), *fscanf*( ), *getchar*( ), *getc*( ), *gets*( ), *scanf*( ), *ungetc*( ), the Base
12450 Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

12451 **CHANGE HISTORY**
12452 First released in Issue 1. Derived from Issue 1 of the SVID.

12453 **Issue 5**
12454 Large File Summit extensions are added.

12455 **Issue 6**
12456 Extensions beyond the ISO C standard are marked.

12457 The following new requirements on POSIX implementations derive from alignment with the
12458 Single UNIX Specification:

12459 • The [EIO] and [EOVERFLOW] mandatory error conditions are added.

12460 • The [ENOMEM] and [ENXIO] optional error conditions are added.

12461 The following changes are made for alignment with the ISO/IEC 9899: 1999 standard:

12462 • The DESCRIPTION is updated to clarify the behavior when the end-of-file indicator for the
12463 input stream is not set.

12464 • The RETURN VALUE section is updated to note that the error indicator shall be set for the
12465 stream.

12466 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/32 is applied, updating the [EAGAIN]
12467 error in the ERRORS section from "the process would be delayed" to "the thread would be
12468 delayed".

12469 **Issue 7**
12470 Austin Group Interpretation 1003.1-2001 #051 is applied, updating the list of functions that mark
12471 the *st_atime* field for update.

fgetpos( )

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 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

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*( ), *re*

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 *fgetpos*( ) is changed for alignment with the ISO/IEC 9899: 1999 standard.

12517 **NAME**

12518 fgets — get a string from a stream

12519 **SYNOPSIS**

12520
```
#include <stdio.h>
```

12521
```
char *fgets(char *restrict s, int n, FILE *restrict stream);
```

12522 **DESCRIPTION**

12523 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12524 conflict between the requirements described here and the ISO C standard is unintentional. This
12525 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

12526 The *fgets*( ) function shall read bytes from *stream* into the array pointed to by *s*, until *n*–1 bytes
12527 are read, or a <newline> is read and transferred to *s*, or an end-of-file condition is encountered.
12528 The string is then terminated with a null byte.

12529 CX The *fgets*( ) function may mark the *st_atime* field of the file associated with *stream* for update. The
12530 *st_atime* field shall be marked for update by the first successful execution of *fgetc*( ), *fgets*( ),
12531 *fread*( ), *fscanf*( ), *getc*( ), *getchar*( ), *gets*( ), or *scanf*( ) using *stream* that returns data not supplied by
12532 a prior call to *ungetc*( ).

12533 **RETURN VALUE**

12534 Upon successful completion, *fgets*( ) shall return *s*. If the stream is at end-of-file, the end-of-file
12535 indicator for the stream shall be set and *fgets*( ) shall return a null pointer. If a read error occurs,
12536 CX the error indicator for the stream shall be set, *fgets*( ) shall return a null pointer, and shall set
12537 *errno* to indicate the error.

12538 **ERRORS**

12539 Refer to *fgetc*( ).

12540 **EXAMPLES**

12541 **Reading Input**

12542 The following example uses *fgets*( ) to read each line of input. {LINE_MAX}, which defines the
12543 maximum size of the input line, is defined in the **<limits.h>** header.

12544
```
#include <stdio.h>
```
12545
```
...
```
12546
```
char line[LINE_MAX];
```
12547
```
...
```
12548
```
while (fgets(line, LINE_MAX, fp) != NULL) {
```
12549
```
...
```
12550
```
}
```
12551
```
...
```

12552 **APPLICATION USAGE**

12553 None.

12554 **RATIONALE**

12555 None.

12556 **FUTURE DIRECTIONS**

12557 None.

12558 **SEE ALSO**

12559 *fgetc( )*, *fopen( )*, *fread( )*, *fscanf( )*, *getc( )*, *getchar( )*, *getdelim( )*, *gets( )*, *scanf( )*, *ungetc( )*, the Base
12560 Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

12561 **CHANGE HISTORY**

12562 First released in Issue 1. Derived from Issue 1 of the SVID.

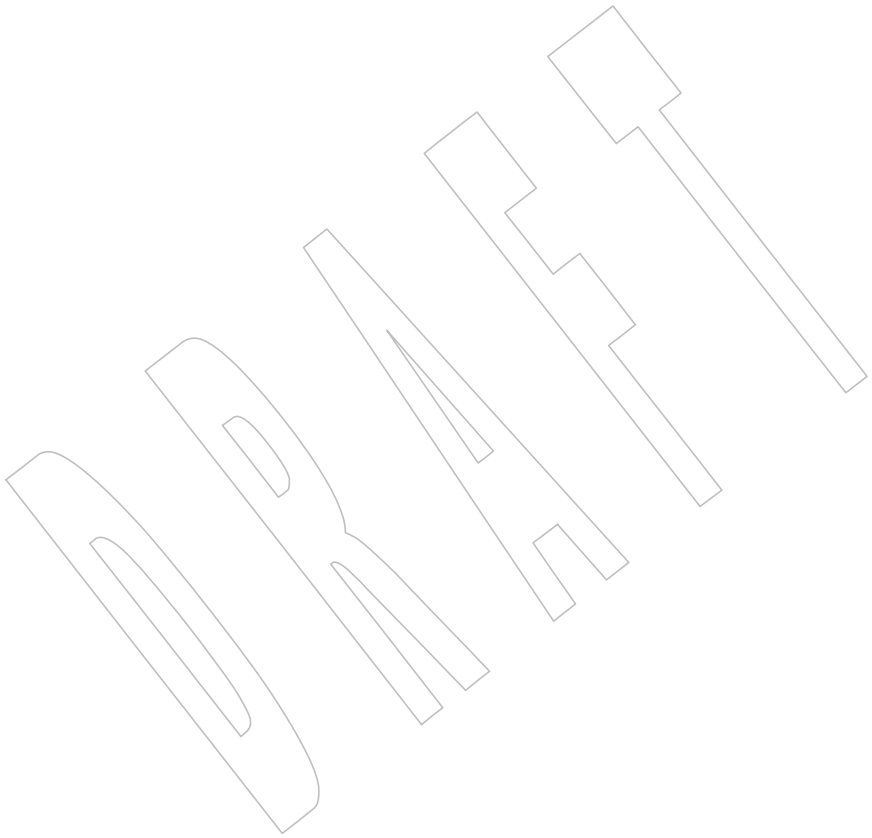
12563 **Issue 6**

12564 Extensions beyond the ISO C standard are marked.

12565 The prototype for *fgets( )* is changed for alignment with the ISO/IEC 9899: 1999 standard.

12566 **Issue 7**

12567 Austin Group Interpretation 1003.1-2001 #051 is applied, updating the list of functions that mark
12568 the *st_atime* field for update.



12569 **NAME**

12570 fgetwc — get a wide-character code from a stream

12571 **SYNOPSIS**

12572 ```
#include <stdio.h>
```

12573 ```
#include <wchar.h>
```

12574 ```
wint_t fgetwc(FILE *stream);
```

12575 **DESCRIPTION**

12576 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12577 conflict between the requirements described here and the ISO C standard is unintentional. This
12578 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

12579 The *fgetwc*( ) function shall obtain the next character (if present) from the input stream pointed to
12580 by *stream*, convert that to the corresponding wide-character code, and advance the associated file
12581 position indicator for the stream (if defined).

12582 If an error occurs, the resulting value of the file position indicator for the stream is unspecified.

12583 CX The *fgetwc*( ) function may mark the *st_atime* field of the file associated with *stream* for update.
12584 The *st_atime* field shall be marked for update by the first successful execution of *fgetc*( ), *fgets*( ),
12585 *fgetwc*( ), *fgetws*( ), *fread*( ), *fscanf*( ), *getc*( ), *getchar*( ), *gets*( ), or *scanf*( ) using *stream* that returns
12586 data not supplied by a prior call to *ungetc*( ) or *ungetwc*( ).

12587 **RETURN VALUE**

12588 Upon successful completion, the *fgetwc*( ) function shall return the wide-character code of the
12589 character read from the input stream pointed to by *stream* converted to a type **wint_t**. If the end-
12590 of-file indicator for the stream is set, or if the stream is at end-of-file, the end-of-file indicator for
12591 the stream shall be set and *fgetwc*( ) shall return WEOF. If a read error occurs, the error indicator
12592 CX for the stream shall be set, *fgetwc*( ) shall return WEOF, and shall set *errno* to indicate the error. If
12593 an encoding error occurs, the error indicator for the stream shall be set, *fgetwc*( ) shall return
12594 WEOF, and shall set *errno* to indicate the error.

12595 **ERRORS**

12596 The *fgetwc*( ) function shall fail if data needs to be read and:

12597 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and
12598 the thread would be delayed in the *fgetwc*( ) operation.

12599 CX [EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for
12600 reading.

12601 [EILSEQ] The data obtained from the input stream does not form a valid character.

12602 CX [EINTR] The read operation was terminated due to the receipt of a signal, and no data
12603 was transferred.

12604 CX [EIO] A physical I/O error has occurred, or the process is in a background process
12605 group attempting to read from its controlling terminal, and either the process
12606 is ignoring or blocking the SIGTTIN signal or the process group is orphaned.
12607 This error may also be generated for implementation-defined reasons.

12608 CX [EOVERFLOW] The file is a regular file and an attempt was made to read at or beyond the
12609 offset maximum associated with the corresponding stream.
```

The *fgetwc*( ) function may fail if:

CX [ENOMEM] Insufficient storage space is available.

CX [ENXIO] 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 1003.1-200x, **<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>**.

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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/33 is applied, updating the [EAGAIN] error in the ERRORS section from "the process would be delayed" to "the thread would be delayed".

Issue 7

Austin Group Interpretation 1003.1-2001 #051 is applied, clarifying the RETURN VALUE section.

12642 **NAME**

12643 fgetws — get a wide-character string from a stream

12644 **SYNOPSIS**

12645 ```
#include <stdio.h>
```

12646 ```
#include <wchar.h>
```

12647 ```
wchar_t *fgetws(wchar_t *restrict ws, int n,
```

12648 ```
    FILE *restrict stream);
```

12649 **DESCRIPTION**

12650 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12651 conflict between the requirements described here and the ISO C standard is unintentional. This
12652 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

12653 The *fgetws*( ) function shall read characters from the *stream*, convert these to the corresponding
12654 wide-character codes, place them in the **wchar_t** array pointed to by *ws*, until *n*–1 characters are
12655 read, or a <newline> is read, converted, and transferred to *ws*, or an end-of-file condition is
12656 encountered. The wide-character string, *ws*, shall then be terminated with a null wide-character
12657 code.

12658 If an error occurs, the resulting value of the file position indicator for the stream is unspecified.

12659 CX The *fgetws*( ) function may mark the *st_atime* field of the file associated with *stream* for update.
12660 The *st_atime* field shall be marked for update by the first successful execution of *fgetc*( ), *fgets*( ),
12661 *fgetwc*( ), *fgetws*( ), *fread*( ), *fscanf*( ), *getc*( ), *getchar*( ), *gets*( ), or *scanf*( ) using *stream* that returns
12662 data not supplied by a prior call to *ungetc*( ) or *ungetwc*( ).

12663 **RETURN VALUE**

12664 Upon successful completion, *fgetws*( ) shall return *ws*. If the end-of-file indicator for the stream is
12665 set, or if the stream is at end-of-file, the end-of-file indicator for the stream shall be set and
12666 *fgetws*( ) shall return a null pointer. If a read error occurs, the error indicator for the stream shall
12667 CX be set, *fgetws*( ) shall return a null pointer, and shall set *errno* to indicate the error.

12668 **ERRORS**

12669 Refer to *fgetwc*( ).

12670 **EXAMPLES**

12671 None.

12672 **APPLICATION USAGE**

12673 None.

12674 **RATIONALE**

12675 None.

12676 **FUTURE DIRECTIONS**

12677 None.

12678 **SEE ALSO**

12679 *fopen*( ), *fread*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**, **<wchar.h>**

12680 **CHANGE HISTORY**

12681 First released in Issue 4. Derived from the MSE working draft.

Issue 5

The Optional Header (OH) marking is removed from **<stdio.h>**.

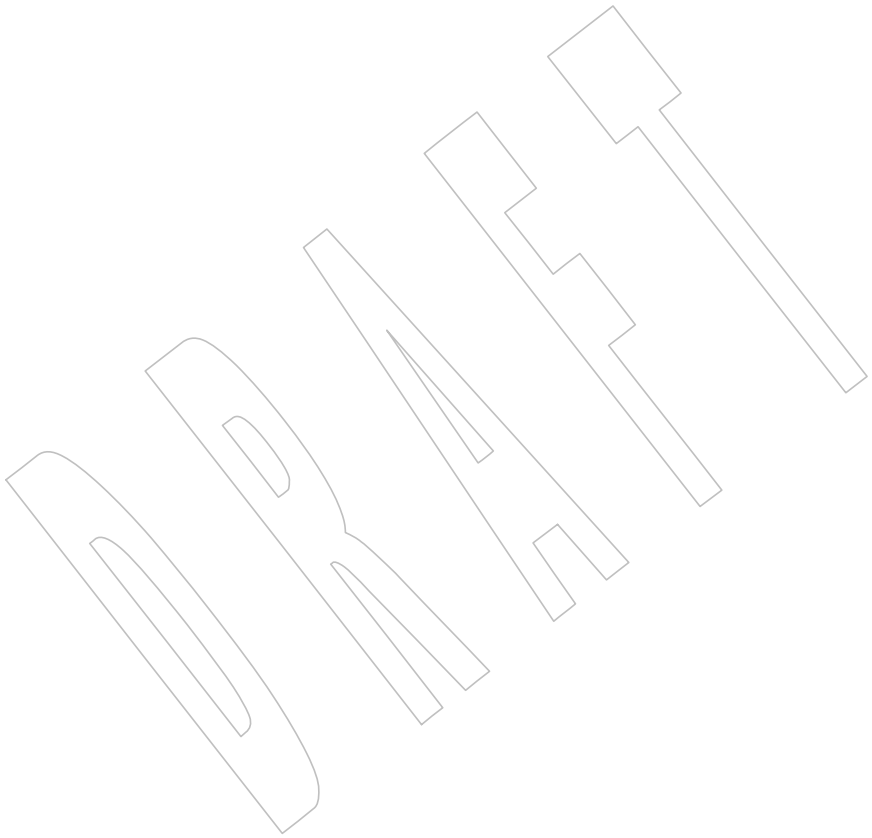
Issue 6

Extensions beyond the ISO C standard are marked.

The prototype for *fgetws*( ) is changed for alignment with the ISO/IEC 9899: 1999 standard.

Issue 7

Austin Group Interpretation 1003.1-2001 #051 is applied, clarifying the RETURN VALUE section.



12689 **NAME**

12690 fileno — map a stream pointer to a file descriptor

12691 **SYNOPSIS**

12692 CX
```
#include <stdio.h>

int fileno(FILE *stream);
```

12694 **DESCRIPTION**

12695 The *fileno*( ) function shall return the integer file descriptor associated with the stream pointed to
12696 by *stream*.

12697 **RETURN VALUE**

12698 Upon successful completion, *fileno*( ) shall return the integer value of the file descriptor
12699 associated with *stream*. Otherwise, the value –1 shall be returned and *errno* set to indicate the
12700 error.

12701 **ERRORS**

12702 The *fileno*( ) function may fail if:

12703 [EBADF] The *stream* argument is not a valid stream.

12704 **EXAMPLES**

12705 None.

12706 **APPLICATION USAGE**

12707 None.

12708 **RATIONALE**

12709 Without some specification of which file descriptors are associated with these streams, it is
12710 impossible for an application to set up the streams for another application it starts with *fork*( )
12711 and *exec*. In particular, it would not be possible to write a portable version of the *sh* command
12712 interpreter (although there may be other constraints that would prevent that portability).

12713 **FUTURE DIRECTIONS**

12714 None.

12715 **SEE ALSO**

12716 Section 2.5.1 (on page 35), *dirfd*( ), *fdopen*( ), *fopen*( ), *stdin*, the Base Definitions volume of
12717 IEEE Std 1003.1-200x, **<stdio.h>**

12718 **CHANGE HISTORY**

12719 First released in Issue 1. Derived from Issue 1 of the SVID.

12720 **Issue 6**

12721 The following new requirements on POSIX implementations derive from alignment with the
12722 Single UNIX Specification:

12723 • The [EBADF] optional error condition is added.

NAME

flockfile, ftrylockfile, funlockfile — stdio locking functions

SYNOPSIS

CX
```
#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 Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

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 1003.1-200x, **<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.

Issue 7

The *flockfile*( ), *ftrylockfile*( ), and *funlockfile*( ) functions are moved from the Thread-Safe Functions option to the Base.

12791 **NAME**

12792 floor, floorf, floorl — floor function

12793 **SYNOPSIS**

12794
```
#include <math.h>
```

12795
```
double floor(double x);
```
12796
```
float floorf(float x);
```
12797
```
long double floorl(long double x);
```

12798 **DESCRIPTION**

12799 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12800 conflict between the requirements described here and the ISO C standard is unintentional. This
12801 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

12802 These functions shall compute the largest integral value not greater than *x*.

12803 An application wishing to check for error situations should set *errno* to zero and call
12804 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
12805 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
12806 zero, an error has occurred.

12807 **RETURN VALUE**

12808 Upon successful completion, these functions shall return the largest integral value not greater
12809 than *x*, expressed as a **double**, **float**, or **long double**, as appropriate for the return type of the
12810 function.

12811 MX If *x* is NaN, a NaN shall be returned.

12812 If *x* is ±0 or ±Inf, *x* shall be returned.

12813 XSI If the correct value would cause overflow, a range error shall occur and *floor*( ), *floorf*( ), and
12814 *floorl*( ) shall return the value of the macro –HUGE_VAL, –HUGE_VALF, and –HUGE_VALL,
12815 respectively.

12816 **ERRORS**

12817 These functions shall fail if:

12818 XSI Range Error — The result would cause an overflow.

12819 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
12820 then *errno* shall be set to [ERANGE]. If the integer expression
12821 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
12822 floating-point exception shall be raised.

12823 **EXAMPLES**

12824 None.

12825 **APPLICATION USAGE**

12826 The integral value returned by these functions might not be expressible as an **int** or **long**. The
12827 return value should be tested before assigning it to an integer type to avoid the undefined
12828 results of an integer overflow.

12829 The *floor*( ) function can only overflow when the floating-point representation has
12830 DBL_MANT_DIG > DBL_MAX_EXP.

12831 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
12832 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

12833 **RATIONALE**

12834 None.

12835 **FUTURE DIRECTIONS**

12836 None.

12837 **SEE ALSO**

12838 *ceil( )*, *feclearexcept( )*, *fetestexcept( )*, *isnan( )*, the Base Definitions volume of IEEE Std 1003.1-200x,
12839 Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

12840 **CHANGE HISTORY**

12841 First released in Issue 1. Derived from Issue 1 of the SVID.

12842 **Issue 5**

12843 The DESCRIPTION is updated to indicate how an application should check for an error. This
12844 text was previously published in the APPLICATION USAGE section.

12845 **Issue 6**

12846 The *floorf( )* and *floorl( )* functions are added for alignment with the ISO/IEC 9899: 1999 standard.

12847 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
12848 revised to align with the ISO/IEC 9899: 1999 standard.

12849 IEC 60559: 1989 standard floating-point extensions over the ISO/IEC 9899: 1999 standard are
12850 marked.

12851 **NAME**

12852 fma, fmaf, fmal — floating-point multiply-add

12853 **SYNOPSIS**

12854
```
#include <math.h>
```

12855
```
double fma(double x, double y, double z);
```
12856
```
float fmaf(float x, float y, float z);
```
12857
```
long double fmal(long double x, long double y, long double z);
```

12858 **DESCRIPTION**

12859 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12860 conflict between the requirements described here and the ISO C standard is unintentional. This
12861 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

12862 These functions shall compute $(x * y) + z$, rounded as one ternary operation: they shall compute
12863 the value (as if) to infinite precision and round once to the result format, according to the
12864 rounding mode characterized by the value of FLT_ROUNDS.

12865 An application wishing to check for error situations should set *errno* to zero and call
12866 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
12867 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
12868 zero, an error has occurred.

12869 **RETURN VALUE**

12870 Upon successful completion, these functions shall return $(x * y) + z$, rounded as one ternary
12871 operation.

12872 MX If the result overflows or underflows, a range error may occur. On systems that support the IEC
12873 60559 Floating-Point option, if the result overflows a range error shall occur.

12874 If x or y are NaN, a NaN shall be returned.

12875 If x multiplied by y is an exact infinity and z is also an infinity but with the opposite sign, a
12876 domain error shall occur, and either a NaN (if supported), or an implementation-defined value
12877 shall be returned.

12878 If one of x and y is infinite, the other is zero, and z is not a NaN, a domain error shall occur, and
12879 either a NaN (if supported), or an implementation-defined value shall be returned.

12880 If one of x and y is infinite, the other is zero, and z is a NaN, a NaN shall be returned and a
12881 domain error may occur.

12882 If $x*y$ is not 0*Inf nor Inf*0 and z is a NaN, a NaN shall be returned.

12883 **ERRORS**

12884 These functions shall fail if:

12885 MX Domain Error The value of $x*y+z$ is invalid, or the value $x*y$ is invalid and z is not a NaN.

12886 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
12887 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
12888 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
12889 shall be raised.

12890 MX Range Error The result overflows.

12891 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
12892 then *errno* shall be set to [ERANGE]. If the integer expression
12893 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow

floating-point exception shall be raised.

These functions may fail if:

MX

Domain Error	The value $x*y$ is invalid and z is a 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.
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.
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

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

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 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

Issue 7

ISO/IEC 9899: 1999 standard, Technical Corrigendum 2 #57 (SD5-XSH-ERN-69) is applied, adding a "may fail" range error for non-MX systems.

12932 **NAME**

12933 fmax, fmaxf, fmaxl — determine maximum numeric value of two floating-point numbers

12934 **SYNOPSIS**

12935
```
#include <math.h>

double fmax(double x, double y);
float fmaxf(float x, float y);
long double fmaxl(long double x, long double y);
```

12939 **DESCRIPTION**

12940 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12941 conflict between the requirements described here and the ISO C standard is unintentional. This
12942 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

12943 MX These functions shall determine the maximum numeric value of their arguments. NaN
12944 arguments shall be treated as missing data: if one argument is a NaN and the other numeric,
12945 then these functions shall choose the numeric value.

12946 **RETURN VALUE**

12947 Upon successful completion, these functions shall return the maximum numeric value of their
12948 arguments.

12949 MX If just one argument is a NaN, the other argument shall be returned.

12950 If *x* and *y* are NaN, a NaN shall be returned.

12951 **ERRORS**

12952 No errors are defined.

12953 **EXAMPLES**

12954 None.

12955 **APPLICATION USAGE**

12956 None.

12957 **RATIONALE**

12958 None.

12959 **FUTURE DIRECTIONS**

12960 None.

12961 **SEE ALSO**

12962 *fdim( )*, *fmin( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

12963 **CHANGE HISTORY**

12964 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

12965 **Issue 7**

12966 Austin Group Interpretation 1003.1-2001 #007 is applied.

12967 **NAME**

12968 fmemopen — open a memory buffer stream

12969 **SYNOPSIS**

12970 CX
```
#include <stdio.h>

FILE *fmemopen(void *restrict buf, size_t size,
    const char *restrict mode);
```

12973 **DESCRIPTION**

12974 The *fmemopen*( ) function shall associate the buffer given by the *buf* and *size* arguments with a
12975 stream. The *buf* argument shall be either a null pointer or point to a buffer that is at least *size*
12976 bytes long.

12977 The *mode* argument is a character string having one of the following values:

12978 *r* or *rb*	Open the stream for reading.
12979 *w* or *wb*	Open the stream for writing.
12980 *a* or *ab*	Append; open the stream for writing at the first null byte.
12981 *r+* or *rb+* or *r+b*	Open the stream for update (reading and writing).
12982 *w+* or *wb+* or *w+b* 12983	Open the stream for update (reading and writing). Truncate the buffer contents.
12984 *a+* or *ab+* or *a+b* 12985	Append; open the stream for update (reading and writing); the initial position is at the first null byte.

12986 The character `'b'` shall have no effect.

12987 If a null pointer is specified as the *buf* argument, *fmemopen*( ) shall allocate *size* bytes of memory
12988 as if by a call to *malloc*( ). This buffer shall be automatically freed when the stream is closed.
12989 Because this feature is only useful when the stream is opened for updating (because there is no
12990 way to get a pointer to the buffer) the *fmemopen*( ) call may fail if the *mode* argument does not
12991 include a `'+'`.

12992 The stream maintains a current position in the buffer. This position is initially set to either the
12993 beginning of the buffer (for *r* and *w* modes) or to the first null byte in the buffer (for *a* modes). If
12994 no null byte is found in append mode, the initial position is set to one byte after the end of the
12995 buffer.

12996 If *buf* is a null pointer, the initial position shall always be set to the beginning of the buffer.

12997 The stream also maintains the size of the current buffer contents. For modes *r* and *r+* the size is
12998 set to the value given by the *size* argument. For modes *w* and *w+* the initial size is zero and for
12999 modes *a* and *a+* the initial size is either the position of the first null byte in the buffer or the value
13000 of the *size* argument if no null byte is found.

13001 A read operation on the stream cannot advance the current buffer position behind the current
13002 buffer size. Reaching the buffer size in a read operation counts as "end-of-file". Null bytes in the
13003 buffer have no special meaning for reads. The read operation starts at the current buffer position
13004 of the stream.

13005 A write operation starts either at the current position of the stream (if mode has not specified
13006 `'a'` as the first character) or at the current size of the stream (if mode had `'a'` as the first
13007 character). If the current position at the end of the write is larger than the current buffer size, the
13008 current buffer size is set to the current position. A write operation on the stream cannot advance

the current buffer size behind the size given in the *size* argument.

When a stream open for writing is flushed or closed, a null byte is written at the current position or at the end of the buffer, depending on the size of the contents. If a stream open for update is flushed or closed and the last write has advanced the current buffer size, a null byte is written at the end of the buffer if it fits.

An attempt to seek a memory buffer stream to a negative position or to a position larger than the buffer size given in the *size* argument shall fail.

RETURN VALUE

Upon successful completion, *fmemopen*( ) 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 *fmemopen*( ) function shall fail if:

[EINVAL] The *size* argument specifies a buffer size of zero.

The *fmemopen*( ) function may fail if:

[EINVAL] The value of the *mode* argument is not valid.

[EINVAL] The *buf* argument is a null pointer and the *mode* argument does not include a `'+'` character.

[ENOMEM] The *buf* argument is a null pointer and the allocation of a buffer of length *size* has failed.

[EMFILE] {FOPEN_MAX} streams are currently open in the calling process.

EXAMPLES

```
#include <stdio.h>

static char buffer[] = "foobar";

int
main (void)
{
    int ch;
    FILE *stream;

    stream = fmemopen(buffer, strlen (buffer), "r");
    if (stream == NULL)
        /* handle error */;

    while ((ch = fgetc(stream)) != EOF)
        printf("Got %c\n", ch);

    fclose(stream);
    return (0);
}
```

This program produces the following output:

```
Got f
Got o
Got o
Got b
Got a
Got r
```


13052 **APPLICATION USAGE**

13053 None.

13054 **RATIONALE**

13055 This interface has been introduced to eliminate many of the errors encountered in the
13056 construction of strings, notably overflowing of strings. This interface prevents overflow.

13057 **FUTURE DIRECTIONS**

13058 None.

13059 **SEE ALSO**

13060 *fdopen*( ), *fopen*( ), *freopen*( ), *malloc*( ), *open_memstream*( ), the Base Definitions volume of
13061 IEEE Std 1003.1-200x, **<stdio.h>**

13062 **CHANGE HISTORY**

13063 First released in Issue 7.

13064 **NAME**

13065 fmin, fminf, fminl — determine minimum numeric value of two floating-point numbers

13066 **SYNOPSIS**

13067
```
#include <math.h>

double fmin(double x, double y);
float fminf(float x, float y);
long double fminl(long double x, long double y);
```

13071 **DESCRIPTION**

13072 CX The functionality described on this reference page is aligned with the ISO C standard. Any
13073 conflict between the requirements described here and the ISO C standard is unintentional. This
13074 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

13075 MX These functions shall determine the minimum numeric value of their arguments. NaN
13076 arguments shall be treated as missing data: if one argument is a NaN and the other numeric,
13077 then these functions shall choose the numeric value.

13078 **RETURN VALUE**

13079 Upon successful completion, these functions shall return the minimum numeric value of their
13080 arguments.

13081 MX If just one argument is a NaN, the other argument shall be returned.

13082 If *x* and *y* are NaN, a NaN shall be returned.

13083 **ERRORS**

13084 No errors are defined.

13085 **EXAMPLES**

13086 None.

13087 **APPLICATION USAGE**

13088 None.

13089 **RATIONALE**

13090 None.

13091 **FUTURE DIRECTIONS**

13092 None.

13093 **SEE ALSO**

13094 *fdim( )*, *fmax( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

13095 **CHANGE HISTORY**

13096 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

13097 **Issue 7**

13098 Austin Group Interpretation 1003.1-2001 #008 is applied.

13099 **NAME**

13100 fmod, fmodf, fmodl — floating-point remainder value function

13101 **SYNOPSIS**

13102
```
#include <math.h>
```

13103
```
double fmod(double x, double y);
```
13104
```
float fmodf(float x, float y);
```
13105
```
long double fmodl(long double x, long double y);
```

13106 **DESCRIPTION**

13107 CX The functionality described on this reference page is aligned with the ISO C standard. Any
13108 conflict between the requirements described here and the ISO C standard is unintentional. This
13109 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

13110 These functions shall return the floating-point remainder of the division of *x* by *y*.

13111 An application wishing to check for error situations should set *errno* to zero and call
13112 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
13113 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
13114 zero, an error has occurred.

13115 **RETURN VALUE**

13116 These functions shall return the value $x-i*y$, for some integer i such that, if y is non-zero, the
13117 result has the same sign as x and magnitude less than the magnitude of y.

13118 If the correct value would cause underflow, and is not representable, a range error may occur,
13119 MX and either 0.0 (if supported), or an implementation-defined value shall be returned.

13120 MX If *x* or *y* is NaN, a NaN shall be returned.

13121 If *y* is zero, a domain error shall occur, and either a NaN (if supported), or an implementation-
13122 defined value shall be returned.

13123 If *x* is infinite, a domain error shall occur, and either a NaN (if supported), or an
13124 implementation-defined value shall be returned.

13125 If *x* is ±0 and *y* is not zero, ±0 shall be returned.

13126 If *x* is not infinite and *y* is ±Inf, *x* shall be returned.

13127 If the correct value would cause underflow, and is representable, a range error may occur and
13128 the correct value shall be returned.

13129 **ERRORS**

13130 These functions shall fail if:

13131 MX Domain Error The *x* argument is infinite or *y* is zero.

13132 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
13133 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
13134 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
13135 shall be raised.

13136 These functions may fail if:

13137 Range Error The result underflows.

13138 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
13139 then *errno* shall be set to [ERANGE]. If the integer expression
13140 (*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

feclearexcept( ), *fetestexcept( )*, *isnan( )*, the Base Definitions volume of IEEE Std 1003.1-200x, 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.

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

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:

MM_HALT	Indicates that the application has encountered a severe fault and is halting. Produces the string `"HALT"`.
MM_ERROR	Indicates that the application has detected a fault. Produces the string `"ERROR"`.
MM_WARNING	Indicates a condition that is out of the ordinary, that might be a problem, and should be watched. Produces the string `"WARNING"`.
MM_INFO	Provides information about a condition that is not in error. Produces the string `"INFO"`.
MM_NOSEV	Indicates that no severity level is supplied for the message.

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.

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.

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"`.

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.

MSGVERB shall determine which components are selected for display to standard error. All message components shall be included in console messages.

RETURN VALUE

The *fmtmsg*( ) function shall return one of the following values:

MM_OK	The function succeeded.
MM_NOTOK	The function failed completely.
MM_NOMSG	The function was unable to generate a message on standard error, but otherwise succeeded.
MM_NOCON	The function was unable to generate a console message, but otherwise succeeded.

ERRORS

None.

EXAMPLES

1. The following example of *fmtmsg*( ):

```
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 1003.1-200x, **<fmtmsg.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

13276 **NAME**

13277 fnmatch — match a filename or a pathname

13278 **SYNOPSIS**

13279 ```
#include <fnmatch.h>
```

13280 ```
int fnmatch(const char *pattern, const char *string, int flags);
```

13281 **DESCRIPTION**

13282 The *fnmatch*( ) function shall match patterns as described in the Shell and Utilities volume of
13283 IEEE Std 1003.1-200x, Section 2.13.1, Patterns Matching a Single Character, and Section 2.13.2,
13284 Patterns Matching Multiple Characters. It checks the string specified by the *string* argument to
13285 see if it matches the pattern specified by the *pattern* argument.

13286 The *flags* argument shall modify the interpretation of *pattern* and *string*. It is the bitwise-
13287 inclusive OR of zero or more of the flags defined in **<fnmatch.h>**. If the FNM_PATHNAME flag
13288 is set in *flags*, then a slash character (`'/'`) in *string* shall be explicitly matched by a slash in
13289 *pattern*; it shall not be matched by either the asterisk or question-mark special characters, nor by
13290 a bracket expression. If the FNM_PATHNAME flag is not set, the slash character shall be treated
13291 as an ordinary character.

13292 If FNM_NOESCAPE is not set in *flags*, a backslash character (`'\'`) in *pattern* followed by any
13293 other character shall match that second character in *string*. In particular, `"\\"` shall match a
13294 backslash in *string*. If FNM_NOESCAPE is set, a backslash character shall be treated as an
13295 ordinary character.

13296 If FNM_PERIOD is set in *flags*, then a leading period (`'.'`) in *string* shall match a period in
13297 *pattern*; as described by rule 2 in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section
13298 2.13.3, Patterns Used for Filename Expansion where the location of "leading" is indicated by the
13299 value of FNM_PATHNAME:

13300 • If FNM_PATHNAME is set, a period is "leading" if it is the first character in *string* or if it
13301 immediately follows a slash.

13302 • If FNM_PATHNAME is not set, a period is "leading" only if it is the first character of
13303 *string*.

13304 If FNM_PERIOD is not set, then no special restrictions are placed on matching a period.

13305 **RETURN VALUE**

13306 If *string* matches the pattern specified by *pattern*, then *fnmatch*( ) shall return 0. If there is no
13307 match, *fnmatch*( ) shall return FNM_NOMATCH, which is defined in **<fnmatch.h>**. If an error
13308 occurs, *fnmatch*( ) shall return another non-zero value.

13309 **ERRORS**

13310 No errors are defined.

13311 **EXAMPLES**

13312 None.

13313 **APPLICATION USAGE**

13314 The *fnmatch*( ) function has two major uses. It could be used by an application or utility that
13315 needs to read a directory and apply a pattern against each entry. The *find* utility is an example of
13316 this. It can also be used by the *pax* utility to process its *pattern* operands, or by applications that
13317 need to match strings in a similar manner.

13318 The name *fnmatch*( ) is intended to imply *filename* match, rather than *pathname* match. The
13319 default action of this function is to match filenames, rather than pathnames, since it gives no
13320 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 at the beginning of a filename.

RATIONALE

This function replaced the REG_FILENAME flag of *regcomp*( ) in early proposals of this volume of IEEE Std 1003.1-200x. 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 1003.1-200x, **<fnmatch.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x

CHANGE HISTORY

First released in Issue 4. Derived from the ISO POSIX-2 standard.

Issue 5

Moved from POSIX2 C-language Binding to BASE.

13338 **NAME**

13339 fopen — open a stream

13340 **SYNOPSIS**

13341 `#include <stdio.h>`

13342 `FILE *fopen(const char *restrict filename, const char *restrict mode);`

13343 **DESCRIPTION**

13344 CX The functionality described on this reference page is aligned with the ISO C standard. Any
13345 conflict between the requirements described here and the ISO C standard is unintentional. This
13346 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

13347 The *fopen*( ) function shall open the file whose pathname is the string pointed to by *filename*, and
13348 associates a stream with it.

13349 The *mode* argument points to a string. If the string is one of the following, the file shall be opened
13350 in the indicated mode. Otherwise, the behavior is undefined.

13351 *r* or *rb* Open file for reading.

13352 *w* or *wb* Truncate to zero length or create file for writing.

13353 *a* or *ab* Append; open or create file for writing at end-of-file.

13354 *r+* or *rb+* or *r+b* Open file for update (reading and writing).

13355 *w+* or *wb+* or *w+b* Truncate to zero length or create file for update.

13356 *a+* or *ab+* or *a+b* Append; open or create file for update, writing at end-of-file.

13357 CX The character `'b'` shall have no effect, but is allowed for ISO C standard conformance. Opening
13358 a file with read mode (*r* as the first character in the *mode* argument) shall fail if the file does not
13359 exist or cannot be read.

13360 Opening a file with append mode (*a* as the first character in the *mode* argument) shall cause all
13361 subsequent writes to the file to be forced to the then current end-of-file, regardless of intervening
13362 calls to *fseek*( ).

13363 When a file is opened with update mode (`'+'` as the second or third character in the *mode*
13364 argument), both input and output may be performed on the associated stream. However, the
13365 application shall ensure that output is not directly followed by input without an intervening call
13366 to *fflush*( ) or to a file positioning function (*fseek*( ), *fsetpos*( ), or *rewind*( )), and input is not directly
13367 followed by output without an intervening call to a file positioning function, unless the input
13368 operation encounters end-of-file.

13369 When opened, a stream is fully buffered if and only if it can be determined not to refer to an
13370 interactive device. The error and end-of-file indicators for the stream shall be cleared.

13371 CX If *mode* is *w*, *wb*, *a*, *ab*, *w+*, *wb+*, *w+b*, *a+*, *ab+*, or *a+b*, and the file did not previously exist, upon
13372 successful completion, the *fopen*( ) function shall mark for update the *st_atime*, *st_ctime*, and
13373 *st_mtime* fields of the file and the *st_ctime* and *st_mtime* fields of the parent directory.

13374 If *mode* is *w*, *wb*, *a*, *ab*, *w+*, *wb+*, *w+b*, *a+*, *ab+*, or *a+b*, and the file did not previously exist, the
13375 *fopen*( ) function shall create a file as if it called the *creat*( ) function with a value appropriate for
13376 the *path* argument interpreted from *filename* and a value of S_IRUSR | S_IWUSR | S_IRGRP |
13377 S_IWGRP | S_IROTH | S_IWOTH for the *mode* argument.

13378 If *mode* is *w*, *wb*, *w+*, *wb+*, or *w+b*, and the file did previously exist, upon successful completion,
13379 *fopen*( ) shall mark for update the *st_ctime* and *st_mtime* fields of the file. The *fopen*( ) function
13380 shall allocate a file descriptor as *open*( ) does.

13381 XSI After a successful call to the *fopen*( ) function, the orientation of the stream shall be cleared, the
13382 encoding rule shall be cleared, and the associated **mbstate_t** object shall be set to describe an
13383 initial conversion state.

13384 CX The largest value that can be represented correctly in an object of type **off_t** shall be established
13385 as the offset maximum in the open file description.

13386 **RETURN VALUE**

13387 Upon successful completion, *fopen*( ) shall return a pointer to the object controlling the stream.
13388 CX Otherwise, a null pointer shall be returned, and *errno* shall be set to indicate the error.

13389 **ERRORS**

13390 The *fopen*( ) function shall fail if:

13391 CX [EACCES] Search permission is denied on a component of the path prefix, or the file
13392 exists and the permissions specified by *mode* are denied, or the file does not
13393 exist and write permission is denied for the parent directory of the file to be
13394 created.

13395 CX [EINTR] A signal was caught during *fopen*( ).

13396 CX [EISDIR] The named file is a directory and *mode* requires write access.

13397 CX [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
13398 argument.

13399 CX [EMFILE] All file descriptors available to the process are currently open.

13400 CX [ENAMETOOLONG]
13401 The length of the *filename* argument exceeds {PATH_MAX} or a pathname
13402 component is longer than {NAME_MAX}.

13403 CX [ENFILE] The maximum allowable number of files is currently open in the system.

13404 CX [ENOENT] A component of *filename* does not name an existing file or *filename* is an empty
13405 string.

13406 CX [ENOSPC] The directory or file system that would contain the new file cannot be
13407 expanded, the file does not exist, and the file was to be created.

13408 CX [ENOTDIR] A component of the path prefix is not a directory.

13409 CX [ENXIO] The named file is a character special or block special file, and the device
13410 associated with this special file does not exist.

13411 CX [EOVERFLOW] The named file is a regular file and the size of the file cannot be represented
13412 correctly in an object of type **off_t**.

13413 CX [EROFS] The named file resides on a read-only file system and *mode* requires write
13414 access.

13415 The *fopen*( ) function may fail if:

13416 CX [EINVAL] The value of the *mode* argument is not valid.

13417 CX [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
13418 resolution of the *path* argument.

13419 CX [EMFILE] {FOPEN_MAX} streams are currently open in the calling process.

13420 CX [EMFILE] {STREAM_MAX} streams are currently open in the calling process.

13421 CX [ENAMETOOLONG]
13422 Pathname resolution of a symbolic link produced an intermediate result
13423 whose length exceeds {PATH_MAX}.

CX [ENOMEM] Insufficient storage space is available.

CX [ETXTBSY] The file is a pure procedure (shared text) file that is being executed and *mode* requires write access.

EXAMPLES

Opening a File

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.

```
#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

creat( ), *fclose*( ), *fdopen*( ), *fmemopen*( ), *freopen*( ), *open_memstream*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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 normative text 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.

Issue 7

SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

Austin Group Interpretation 1003.1-2001 #025 is applied, clarifying the file creation mode.

13477 **NAME**

13478 fork — create a new process

13479 **SYNOPSIS**

13480
```
#include <unistd.h>
```

13481
```
pid_t fork(void);
```

13482 **DESCRIPTION**

13483 The *fork*( ) function shall create a new process. The new process (child process) shall be an exact
13484 copy of the calling process (parent process) except as detailed below:

13485 • The child process shall have a unique process ID.

13486 • The child process ID also shall not match any active process group ID.

13487 • The child process shall have a different parent process ID, which shall be the process ID of
13488 the calling process.

13489 • The child process shall have its own copy of the parent's file descriptors. Each of the
13490 child's file descriptors shall refer to the same open file description with the corresponding
13491 file descriptor of the parent.

13492 • The child process shall have its own copy of the parent's open directory streams. Each
13493 open directory stream in the child process may share directory stream positioning with the
13494 corresponding directory stream of the parent.

13495 • The child process shall have its own copy of the parent's message catalog descriptors.

13496 • The child process values of *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* shall be set to
13497 0.

13498 • The time left until an alarm clock signal shall be reset to zero, and the alarm, if any, shall be
13499 canceled; see *alarm*( ).

13500 XSI • All *semadj* values shall be cleared.

13501 • File locks set by the parent process shall not be inherited by the child process.

13502 • The set of signals pending for the child process shall be initialized to the empty set.

13503 XSI • Interval timers shall be reset in the child process.

13504 • Any semaphores that are open in the parent process shall also be open in the child process.

13505 ML • The child process shall not inherit any address space memory locks established by the
13506 parent process via calls to *mlockall*( ) or *mlock*( ).

13507 • Memory mappings created in the parent shall be retained in the child process.
13508 MAP_PRIVATE mappings inherited from the parent shall also be MAP_PRIVATE
13509 mappings in the child, and any modifications to the data in these mappings made by the
13510 parent prior to calling *fork*( ) shall be visible to the child. Any modifications to the data in
13511 MAP_PRIVATE mappings made by the parent after *fork*( ) returns shall be visible only to
13512 the parent. Modifications to the data in MAP_PRIVATE mappings made by the child shall
13513 be visible only to the child.

13514 PS • For the SCHED_FIFO and SCHED_RR scheduling policies, the child process shall inherit
13515 the policy and priority settings of the parent process during a *fork*( ) function. For other
13516 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.

MSG

- 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 description as the corresponding message descriptor of the parent.

- No asynchronous input or asynchronous output operations shall be inherited by the child process. Any use of asynchronous control blocks created by the parent produces undefined behavior.

- 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.

  When the application calls *fork*( ) from a signal handler and any of the fork handlers registered by *pthread_atfork*( ) calls a function that is not asynch-signal-safe, the behavior is undefined.

OB 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.

OB 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.

OB 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 1003.1-200x shall be the same in the parent and child processes. The inheritance of process characteristics not defined by IEEE Std 1003.1-200x is unspecified by IEEE Std 1003.1-200x.

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.

The *fork*( ) function may fail if:

[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

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*( ).

While the *fork*( ) function is async-signal-safe, there is no way for an implementation to determine whether the fork handlers established by *pthread_atfork*( ) are async-signal-safe. The fork handlers may attempt to execute portions of the implementation that are not async-signal-safe, such as those that are protected by mutexes, leading to a deadlock condition. It is therefore undefined for the fork handlers to execute functions that are not async-signal-safe when *fork*( ) is called from a signal handler.

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*( ), *pthread_atfork*( ), *semop*( ), *signal*( ), *times*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.10, Memory Synchronization, **<sys/types.h>**, **<unistd.h>**

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 required for UNIX applications.

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.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/17 is applied, adding text to the DESCRIPTION and RATIONALE relating to fork handlers registered by the *pthread_atfork*( ) function and async-signal safety.

Issue 7

Austin Group Interpretation 1003.1-2001 #080 is applied, clarifying the status of asynchronous input and asynchronous output operations and asynchronous control lists in the DESCRIPTION.

Functionality relating to the Asynchronous Input and Output, Memory Mapped Files, Timers, and Threads options is moved to the Base.

Functionality relating to message catalog descriptors is moved from the XSI option to the Base.

NAME

fpathconf, pathconf — get configurable pathname variables

SYNOPSIS

```
#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*.

Variable	Value of *name*	Requirements
{FILESIZEBITS}	_PC_FILESIZEBITS	3,4
{LINK_MAX}	_PC_LINK_MAX	1
{MAX_CANON}	_PC_MAX_CANON	2
{MAX_INPUT}	_PC_MAX_INPUT	2
{NAME_MAX}	_PC_NAME_MAX	3,4
{PATH_MAX}	_PC_PATH_MAX	4,5
{PIPE_BUF}	_PC_PIPE_BUF	6
{POSIX2_SYMLINKS}	_PC_2_SYMLINKS	4
{POSIX_ALLOC_SIZE_MIN}	_PC_ALLOC_SIZE_MIN	10
{POSIX_REC_INCR_XFER_SIZE}	_PC_REC_INCR_XFER_SIZE	10
{POSIX_REC_MAX_XFER_SIZE}	_PC_REC_MAX_XFER_SIZE	10
{POSIX_REC_MIN_XFER_SIZE}	_PC_REC_MIN_XFER_SIZE	10
{POSIX_REC_XFER_ALIGN}	_PC_REC_XFER_ALIGN	10
{SYMLINK_MAX}	_PC_SYMLINK_MAX	4,9
_POSIX_CHOWN_RESTRICTED	_PC_CHOWN_RESTRICTED	7
_POSIX_NO_TRUNC	_PC_NO_TRUNC	3,4
_POSIX_VDISABLE	_PC_VDISABLE	2
_POSIX_ASYNC_IO	_PC_ASYNC_IO	8
_POSIX_PRIO_IO	_PC_PRIO_IO	8
_POSIX_SYNC_IO	_PC_SYNC_IO	8

Requirements

1. If *path* or *fildes* refers to a directory, the value returned shall apply to the directory itself.
2. If *path* or *fildes* does not refer to a terminal file, it is unspecified whether an implementation supports an association of the variable name with the specified file.
3. If *path* or *fildes* refers to a directory, the value returned shall apply to filenames within the directory.
4. If *path* or *fildes* does not refer to a directory, it is unspecified whether an implementation supports an association of the variable name with the specified file.

5. If *path* or *fildes* refers to a directory, the value returned shall be the maximum length of a relative pathname when the specified directory is the working directory.

6. If *path* refers to a FIFO, or *fildes* refers to a pipe or FIFO, the value returned shall apply to the referenced object. If *path* or *fildes* refers to a directory, the value returned shall apply to any FIFO that exists or can be created within the directory. If *path* or *fildes* refers to any other type of file, it is unspecified whether an implementation supports an association of the variable name with the specified file.

7. If *path* or *fildes* refers to a directory, the value returned shall apply to any files, other than directories, that exist or can be created within the directory.

8. If *path* or *fildes* refers to a directory, it is unspecified whether an implementation supports an association of the variable name with the specified file.

9. If *path* or *fildes* refers to a directory, the value returned shall be the maximum length of the string that a symbolic link in that directory can contain.

10. If *path* or *fildes* des does not refer to a regular file, it is unspecified whether an implementation supports an association of the variable name with the specified file. If an implementation supports such an association for other than a regular file, the value returned is unspecified.

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>**.

If the variable corresponding to *name* is dependent on an unsupported option, the results are unspecified.

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:

[EACCES] Search permission is denied for a component of the path prefix.

[EINVAL] The implementation does not support an association of the variable *name* with the specified file.

[ELOOP] More than {SYMLOOP_MAX} symbolic links were 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}.

[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}.

[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 *fpathconf*( ) function shall fail if:

[EINVAL] The value of *name* is not valid.

The *fpathconf*( ) function may fail if:

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

[EINVAL] The implementation does not support an association of the variable *name* with the specified file.

EXAMPLES

None.

APPLICATION USAGE

Application writers should check whether an option, such as _POSIX_ADVISORY_INFO, is supported prior to obtaining and using values for related variables such as {POSIX_ALLOC_SIZE_MIN}.

RATIONALE

The *pathconf*( ) function was proposed immediately after the *sysconf*( ) function when it was realized that some configurable values may differ across file system, directory, or device boundaries.

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 current working directory of the process. 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 lifetime of the process. 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.

It was the intention of IEEE Std 1003.1d-1999 that the following variables:

{POSIX_ALLOC_SIZE_MIN}
{POSIX_REC_INCR_XFER_SIZE}
{POSIX_REC_MAX_XFER_SIZE}
{POSIX_REC_MIN_XFER_SIZE}
{POSIX_REC_XFER_ALIGN}

only applied to regular files, but Note 10 also permits implementation of the advisory semantics on other file types unique to an implementation (for eample, a character special device).

FUTURE DIRECTIONS

None.

SEE ALSO

chown( ), *confstr*( ), *sysconf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<limits.h>**, **<unistd.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x, *getconf*

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}

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/18 is applied, changing the fourth paragraph of the DESCRIPTION and removing shading and margin markers from the table. This change is needed since implementations are required to support all of these symbols.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/34 is applied, adding the table entry for POSIX2_SYMLINKS in the DESCRIPTION.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/35 is applied, updating the DESCRIPTION and RATIONALE sections to clarify behavior for the {POSIX_ALLOC_SIZE_MIN}, {POSIX_REC_INCR_XFER_SIZE}, {POSIX_REC_MAX_XFER_SIZE}, {POSIX_REC_MIN_XFER_SIZE}, and {POSIX_REC_XFER_ALIGN} variables.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/36 is applied, updating the RETURN VALUE and APPLICATION USAGE sections to state that the results are unspecified if a variable is dependent on an unsupported option, and advising application writers to check for supported options prior to obtaining and using such values.

13873 **NAME**

13874 fpclassify — classify real floating type

13875 **SYNOPSIS**

13876
```
#include <math.h>
```

13877
```
int fpclassify(real-floating x);
```

13878 **DESCRIPTION**

13879 CX The functionality described on this reference page is aligned with the ISO C standard. Any
13880 conflict between the requirements described here and the ISO C standard is unintentional. This
13881 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

13882 The *fpclassify*( ) macro shall classify its argument value as NaN, infinite, normal, subnormal,
13883 zero, or into another implementation-defined category. First, an argument represented in a
13884 format wider than its semantic type is converted to its semantic type. Then classification is based
13885 on the type of the argument.

13886 **RETURN VALUE**

13887 The *fpclassify*( ) macro shall return the value of the number classification macro appropriate to
13888 the value of its argument.

13889 **ERRORS**

13890 No errors are defined.

13891 **EXAMPLES**

13892 None.

13893 **APPLICATION USAGE**

13894 None.

13895 **RATIONALE**

13896 None.

13897 **FUTURE DIRECTIONS**

13898 None.

13899 **SEE ALSO**

13900 *isfinite*( ), *isinf*( ), *isnan*( ), *isnormal*( ), *signbit*( ), the Base Definitions volume of
13901 IEEE Std 1003.1-200x, **<math.h>**

13902 **CHANGE HISTORY**

13903 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

13904 **NAME**

13905 dprintf, fprintf, printf, snprintf, sprintf — print formatted output

13906 **SYNOPSIS**

13907
```
#include <stdio.h>
```

13908 CX
```
int dprintf(int fildes, const char *format, ...);
```
13909
```
int fprintf(FILE *restrict stream, const char *restrict format, ...);
```
13910
```
int printf(const char *restrict format, ...);
```
13911
```
int snprintf(char *restrict s, size_t n,
    const char *restrict format, ...);
```
13912
13913
```
int sprintf(char *restrict s, const char *restrict format, ...);
```

13914 **DESCRIPTION**

13915 CX Excluding *dprintf*( ): The functionality described on this reference page is aligned with the ISO C
13916 standard. Any conflict between the requirements described here and the ISO C standard is
13917 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

13918 The *fprintf*( ) function shall place output on the named output *stream*. The *printf*( ) function shall
13919 place output on the standard output stream *stdout*. The *sprintf*( ) function shall place output
13920 followed by the null byte, `'\0'`, in consecutive bytes starting at *s; it is the user's responsibility
13921 to ensure that enough space is available.

13922 CX The *dprintf*( ) function shall be equivalent to the *fprintf*( ) function, except that *dprintf*( ) shall
13923 write output to the file associated with the file descriptor specified by the *fildes* argument rather
13924 than place output on a stream.

13925 The *snprintf*( ) function shall be equivalent to *sprintf*( ), with the addition of the *n* argument
13926 which states the size of the buffer referred to by *s*. If *n* is zero, nothing shall be written and *s*
13927 may be a null pointer. Otherwise, output bytes beyond the *n*-1st shall be discarded instead of
13928 being written to the array, and a null byte is written at the end of the bytes actually written into
13929 the array.

13930 If copying takes place between objects that overlap as a result of a call to *sprintf*( ) or *snprintf*( ),
13931 the results are undefined.

13932 Each of these functions converts, formats, and prints its arguments under control of the *format*.
13933 The *format* is a character string, beginning and ending in its initial shift state, if any. The *format* is
13934 composed of zero or more directives: *ordinary characters*, which are simply copied to the output
13935 stream, and *conversion specifications*, each of which shall result in the fetching of zero or more
13936 arguments. The results are undefined if there are insufficient arguments for the *format*. If the
13937 *format* is exhausted while arguments remain, the excess arguments shall be evaluated but are
13938 otherwise ignored.

13939 CX Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than
13940 to the next unused argument. In this case, the conversion specifier character % (see below) is
13941 replaced by the sequence "%*n*$", where *n* is a decimal integer in the range [1,{NL_ARGMAX}],
13942 giving the position of the argument in the argument list. This feature provides for the definition
13943 of format strings that select arguments in an order appropriate to specific languages (see the
13944 EXAMPLES section).

13945 The *format* can contain either numbered argument conversion specifications (that is, "%*n*$" and
13946 "**m*$"), or unnumbered argument conversion specifications (that is, % and *), but not both. The
13947 only exception to this is that %% can be mixed with the "%*n*$" form. The results of mixing
13948 numbered and unnumbered argument specifications in a *format* string are undefined. When
13949 numbered argument specifications are used, specifying the *N*th argument requires that all the
13950 leading arguments, from the first to the (*N*–*1*)th, are specified in the format string.

13951 In format strings containing the "%*n*$" form of conversion specification, numbered arguments
13952 in the argument list can be referenced from the format string as many times as required.

13953 In format strings containing the % form of conversion specification, each conversion specification
13954 uses the first unused argument in the argument list.

13955 CX All forms of the *fprintf*() functions allow for the insertion of a language-dependent radix
13956 character in the output string. The radix character is defined in the process' locale (category
13957 *LC_NUMERIC*). In the POSIX locale, or in a locale where the radix character is not defined, the
13958 radix character shall default to a period ('.').

13959 CX Each conversion specification is introduced by the '%' character or by the character sequence
13960 "%*n*$", after which the following appear in sequence:

13961 • Zero or more *flags* (in any order), which modify the meaning of the conversion
13962 specification.

13963 • An optional minimum *field width*. If the converted value has fewer bytes than the field
13964 width, it shall be padded with spaces by default on the left; it shall be padded on the right
13965 if the left-adjustment flag ('–'), described below, is given to the field width. The field
13966 width takes the form of an asterisk ('*'), described below, or a decimal integer.

13967 • An optional *precision* that gives the minimum number of digits to appear for the d, i, o, u,
13968 x, and X conversion specifiers; the number of digits to appear after the radix character for
13969 the a, A, e, E, f, and F conversion specifiers; the maximum number of significant digits for
13970 the g and G conversion specifiers; or the maximum number of bytes to be printed from a
13971 XSI string in the s and S conversion specifiers. The precision takes the form of a period ('.')
13972 followed either by an asterisk ('*'), described below, or an optional decimal digit string,
13973 where a null digit string is treated as zero. If a precision appears with any other conversion
13974 specifier, the behavior is undefined.

13975 • An optional length modifier that specifies the size of the argument.

13976 • A *conversion specifier* character that indicates the type of conversion to be applied.

13977 A field width, or precision, or both, may be indicated by an asterisk ('*'). In this case an
13978 argument of type **int** supplies the field width or precision. Applications shall ensure that
13979 arguments specifying field width, or precision, or both appear in that order before the argument,
13980 if any, to be converted. A negative field width is taken as a '–' flag followed by a positive field
13981 CX width. A negative precision is taken as if the precision were omitted. In *format* strings
13982 containing the "%*n*$" form of a conversion specification, a field width or precision may be
13983 indicated by the sequence "**m*$", where *m* is a decimal integer in the range [1,{NL_ARGMAX}]
13984 giving the position in the argument list (after the *format* argument) of an integer argument
13985 containing the field width or precision, for example:

13986 ```
printf("%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```

13987 The flag characters and their meanings are:

13988 CX ' The integer portion of the result of a decimal conversion (%i, %d, %u, %f, %F, %g, or %G)
13989 shall be formatted with thousands' grouping characters. For other conversions the
13990 behavior is undefined. The non-monetary grouping character is used.

13991 – The result of the conversion shall be left-justified within the field. The conversion is
13992 right-justified if this flag is not specified.

13993 + The result of a signed conversion shall always begin with a sign ('+' or '–'). The
13994 conversion shall begin with a sign only when a negative value is converted if this flag is
13995 not specified.

  
```

13996 <space> If the first character of a signed conversion is not a sign or if a signed conversion results
13997 in no characters, a <space> shall be prefixed to the result. This means that if the
13998 <space> and '+' flags both appear, the <space> flag shall be ignored.

13999 # Specifies that the value is to be converted to an alternative form. For o conversion, it
14000 increases the precision (if necessary) to force the first digit of the result to be zero. For x
14001 or X conversion specifiers, a non-zero result shall have 0x (or 0X) prefixed to it. For a, A,
14002 e, E, f, F, g, and G conversion specifiers, the result shall always contain a radix
14003 character, even if no digits follow the radix character. Without this flag, a radix
14004 character appears in the result of these conversions only if a digit follows it. For g and G
14005 conversion specifiers, trailing zeros shall *not* be removed from the result as they
14006 normally are. For other conversion specifiers, the behavior is undefined.

14007 0 For d, i, o, u, x, X, a, A, e, E, f, F, g, and G conversion specifiers, leading zeros
14008 (following any indication of sign or base) are used to pad to the field width; no space
14009 padding is performed. If the '0' and '−' flags both appear, the '0' flag is ignored.
14010 For d, i, o, u, x, and X conversion specifiers, if a precision is specified, the '0' flag
14011 CX shall be ignored. If the '0' and ''' flags both appear, the grouping characters are
14012 inserted before zero padding. For other conversions, the behavior is undefined.

14013 The length modifiers and their meanings are:

14014 hh Specifies that a following d, i, o, u, x, or X conversion specifier applies to a **signed char**
14015 or **unsigned char** argument (the argument will have been promoted according to the
14016 integer promotions, but its value shall be converted to **signed char** or **unsigned char**
14017 before printing); or that a following n conversion specifier applies to a pointer to a
14018 **signed char** argument.

14019 h Specifies that a following d, i, o, u, x, or X conversion specifier applies to a **short** or
14020 **unsigned short** argument (the argument will have been promoted according to the
14021 integer promotions, but its value shall be converted to **short** or **unsigned short** before
14022 printing); or that a following n conversion specifier applies to a pointer to a **short**
14023 argument.

14024 l (ell) Specifies that a following d, i, o, u, x, or X conversion specifier applies to a **long** or
14025 **unsigned long** argument; that a following n conversion specifier applies to a pointer to
14026 a **long** argument; that a following c conversion specifier applies to a **wint_t** argument;
14027 that a following s conversion specifier applies to a pointer to a **wchar_t** argument; or
14028 has no effect on a following a, A, e, E, f, F, g, or G conversion specifier.

14029 ll (ell-ell)
14030 Specifies that a following d, i, o, u, x, or X conversion specifier applies to a **long long**
14031 or **unsigned long long** argument; or that a following n conversion specifier applies to a
14032 pointer to a **long long** argument.

14033 j Specifies that a following d, i, o, u, x, or X conversion specifier applies to an **intmax_t**
14034 or **uintmax_t** argument; or that a following n conversion specifier applies to a pointer
14035 to an **intmax_t** argument.

14036 z Specifies that a following d, i, o, u, x, or X conversion specifier applies to a **size_t** or the
14037 corresponding signed integer type argument; or that a following n conversion specifier
14038 applies to a pointer to a signed integer type corresponding to a **size_t** argument.

14039 t Specifies that a following d, i, o, u, x, or X conversion specifier applies to a **ptrdiff_t** or
14040 the corresponding **unsigned** type argument; or that a following n conversion specifier
14041 applies to a pointer to a **ptrdiff_t** argument.

14042 L Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to a **long**
14043 **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. If a radix character appears, at least one digit appears before it. The low-order digit shall be rounded in an implementation-defined manner.

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(`*n-char-sequence*`)`" or "`[−]nan`"; 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.

`e`, `E` The **double** argument shall be converted in the style "`[−]`*d*`.`*ddd*`e±`*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.

A **double** argument representing an infinity or NaN shall be converted in the style of an `f` or `F` conversion specifier.

14091 g, G The **double** argument representing a floating-point number shall be converted in the
14092 style f or e (or in the style F or E in the case of a G conversion specifier), depending on
14093 the value converted and the precision. Let P equal the precision if non-zero, 6 if the
14094 precision is omitted, or 1 if the precision is zero. Then, if a conversion with style E
14095 would have an exponent of *X*:

14096 — If P>*X*>=–4, the conversion shall be with style f (or F) and precision P–(*X*+1).

14097 — Otherwise, the conversion shall be with style e (or E) and precision P–1.

14098 Finally, unless the '#' flag is used, any trailing zeros shall be removed from the
14099 fractional portion of the result and the decimal-point character shall be removed if there
14100 is no fractional portion remaining.

14101 A **double** argument representing an infinity or NaN shall be converted in the style of
14102 an f or F conversion specifier.

14103 a, A A **double** argument representing a floating-point number shall be converted in the
14104 style "[–]0x*h*.*hhhh*p±*d*", where there is one hexadecimal digit (which shall be non-
14105 zero if the argument is a normalized floating-point number and is otherwise
14106 unspecified) before the decimal-point character and the number of hexadecimal digits
14107 after it is equal to the precision; if the precision is missing and FLT_RADIX is a power
14108 of 2, then the precision shall be sufficient for an exact representation of the value; if the
14109 precision is missing and FLT_RADIX is not a power of 2, then the precision shall be
14110 sufficient to distinguish values of type **double**, except that trailing zeros may be
14111 omitted; if the precision is zero and the '#' flag is not specified, no decimal-point
14112 character shall appear. The letters "abcdef" shall be used for a conversion and the
14113 letters "ABCDEF" for A conversion. The A conversion specifier produces a number with
14114 'X' and 'P' instead of 'x' and 'p'. The exponent shall always contain at least one
14115 digit, and only as many more digits as necessary to represent the decimal exponent of
14116 2. If the value is zero, the exponent shall be zero.

14117 A **double** argument representing an infinity or NaN shall be converted in the style of
14118 an f or F conversion specifier.

14119 c The **int** argument shall be converted to an **unsigned char**, and the resulting byte shall
14120 be written.

14121 If an l (ell) qualifier is present, the **wint_t** argument shall be converted as if by an ls
14122 conversion specification with no precision and an argument that points to a two-
14123 element array of type **wchar_t**, the first element of which contains the **wint_t** argument
14124 to the ls conversion specification and the second element contains a null wide
14125 character.

14126 s The argument shall be a pointer to an array of **char**. Bytes from the array shall be
14127 written up to (but not including) any terminating null byte. If the precision is specified,
14128 no more than that many bytes shall be written. If the precision is not specified or is
14129 greater than the size of the array, the application shall ensure that the array contains a
14130 null byte.

14131 If an l (ell) qualifier is present, the argument shall be a pointer to an array of type
14132 **wchar_t**. Wide characters from the array shall be converted to characters (each as if by
14133 a call to the *wcrtomb*( ) function, with the conversion state described by an **mbstate_t**
14134 object initialized to zero before the first wide character is converted) up to and
14135 including a terminating null wide character. The resulting characters shall be written
14136 up to (but not including) the terminating null character (byte). If no precision is
14137 specified, the application shall ensure that the array contains a null wide character. If a
14138 precision is specified, no more than that many characters (bytes) shall be written
14139 (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.

`p` The argument shall be a pointer to **void**. The value of the pointer is converted to a sequence of printable characters, in an implementation-defined manner.

`n` The argument shall be a pointer to an integer into which is written the number of bytes written to the output so far by this call to one of the *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.

CX 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

CX Upon successful completion, the *dprintf*( ), *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.

ERRORS

CX For the conditions under which *dprintf*( ), *fprintf*( ), and *printf*( ) fail and may fail, refer to *fputc*( ) or *fputwc*( ).

In addition, all forms of *fprintf*( ) may fail if:

CX [EILSEQ] A wide-character code that does not correspond to a valid character has been detected.

CX [EINVAL] There are insufficient arguments.

The *dprintf*( ) function may fail if:

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

CX The *dprintf*( ), *fprintf*( ), and *printf*( ) functions may fail if:

CX [ENOMEM] Insufficient storage space is available.

The *snprintf*( ) function shall fail if:

CX [EOVERFLOW] The value of *n* is greater than {INT_MAX} or the number of bytes needed to hold the output excluding the terminating null is greater than {INT_MAX}.

EXAMPLES

Printing Language-Independent Date and Time

The following statement can be used to print date and time using a language-independent format:

```
printf(format, weekday, month, day, hour, min);
```

For American usage, *format* could be a pointer to the following string:

```
"%s, %s %d, %d:%.2d\n"
```

This example would produce the following message:

```
Sunday, July 3, 10:02
```

For German usage, *format* could be a pointer to the following string:

```
"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"
```

This definition of *format* would produce the following message:

```
Sonntag, 3. Juli, 10:02
```

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 *printf*( ) use data decoded from a previous *stat*( ) call. The user-defined *strperm*( ) function shall return a string similar to the one at the beginning of the output for the following command:

```
ls –l
```

The next call to *printf*( ) outputs the owner's name if it is found using *getpwuid*( ); the *getpwuid*( ) function shall return a **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 *getgrgid*( ); *getgrgid*( ) is very similar to *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.

```
#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.

```
#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**.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

#define LOCKFILE "/etc/ptmp"
#define PASSWDFILE "/etc/passwd"
...
int pfd;
...
if ((pfd = open(LOCKFILE, O_WRONLY | O_CREAT | O_EXCL,
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
{
    fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n");
    exit(1);
}
...
if (link(LOCKFILE,PASSWDFILE) == -1) {
    fprintf(stderr, "Link error: %s\n", strerror(errno));
    exit(1);
}
...
```

Printing Usage Information

The following example checks to make sure the program has the necessary arguments, and uses *fprintf*( ) to print usage information if the expected number of arguments is not present.

```
#include <stdio.h>
#include <stdlib.h>
...
char *Options = "hdbtl";
...
if (argc < 2) {
    fprintf(stderr, "Usage: %s -%s <file\n", argv[0], Options); exit(1);
}
...
```

Formatting a Decimal String

The following example prints a key and data pair on *stdout*. Note use of the '*' (asterisk) in the format string; this ensures the correct number of decimal places for the element based on the number of elements requested.

```
#include <stdio.h>
...
long i;
char *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.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
...
char filename[PATH_MAX+1];
struct passwd *pw;
...
sprintf(filename, "%s/%d.out", pw->pw_dir, getpid());
...
```

Reporting an Event

The following example loops until an event has timed out. The *pause*( ) function waits forever unless it receives a signal. The *fprintf*( ) statement should never occur due to the possible return values of *pause*( ).

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
...
while (!event_complete) {
...
    if (pause() != –1 || errno != EINTR)
        fprintf(stderr, "pause: unknown error: %s\n", strerror(errno));
}
...
```


Printing Monetary Information

The following example uses *strfmon*( ) to convert a number and store it as a formatted monetary string named *convbuf*. If the first number is printed, the program prints the format and the description; otherwise, it just prints the number.

```
#include <monetary.h>
#include <stdio.h>
...
struct tblfmt {
    char *format;
    char *description;
};

struct tblfmt table[] = {
    { "%n", "default formatting" },
    { "%11n", "right align within an 11 character field" },
    { "%#5n", "aligned columns for values up to 99999" },
    { "%=*#5n", "specify a fill character" },
    { "%=0#5n", "fill characters do not use grouping" },
    { "%^#5n", "disable the grouping separator" },
    { "%^#5.0n", "round off to whole units" },
    { "%^#5.4n", "increase the precision" },
    { "%(#5n", "use an alternative pos/neg style" },
    { "%!(#5n", "disable the currency symbol" },
};
...
float input[3];
int i, j;
char convbuf[100];
...
strfmon(convbuf, sizeof(convbuf), table[i].format, input[j]);

if (j == 0) {
    printf("%s%s%s\n", table[i].format,
        convbuf, table[i].description);
}
else {
    printf("%s\n", convbuf);
}
...
```

Printing Wide Characters

The following example prints a series of wide characters. Suppose that `"L‘@‘"` expands to three bytes:

```
wchar_t wz [3] = L"@@";       // Zero-terminated
wchar_t wn [3] = L"@@@";      // Unterminated

fprintf (stdout,"%ls", wz);   // Outputs 6 bytes
fprintf (stdout,"%ls", wn);   // Undefined because wn has no terminator
fprintf (stdout,"%4ls", wz);  // Outputs 3 bytes
fprintf (stdout,"%4ls", wn);  // Outputs 3 bytes; no terminator needed
fprintf (stdout,"%9ls", wz);  // Outputs 6 bytes
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 1003.1-200x, 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 normative text 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.

ISO/IEC 9899: 1999 standard, Technical Corrigendum 1 is incorporated.

An example of printing wide characters is added.

Issue 7

ISO/IEC 9899: 1999 standard, Technical Corrigendum 2 #68 (SD5-XSH-ERN-70) is applied, revising the description of `g` and `G`.

The *dprintf*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 1.

Functionality relating to the `%n$` form of conversion specification and the `'''` (apostrophe) flag is moved from the XSI option to the Base.

14444 **NAME**

14445 fputc — put a byte on a stream

14446 **SYNOPSIS**

14447
```
#include <stdio.h>
```

14448
```
int fputc(int c, FILE *stream);
```

14449 **DESCRIPTION**

14450 CX The functionality described on this reference page is aligned with the ISO C standard. Any
14451 conflict between the requirements described here and the ISO C standard is unintentional. This
14452 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

14453 The *fputc*( ) function shall write the byte specified by *c* (converted to an **unsigned char**) to the
14454 output stream pointed to by *stream*, at the position indicated by the associated file-position
14455 indicator for the stream (if defined), and shall advance the indicator appropriately. If the file
14456 cannot support positioning requests, or if the stream was opened with append mode, the byte
14457 shall be appended to the output stream.

14458 CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful
14459 execution of *fputc*( ) and the next successful completion of a call to *fflush*( ) or *fclose*( ) on the same
14460 stream or a call to *exit*( ) or *abort*( ).

14461 **RETURN VALUE**

14462 Upon successful completion, *fputc*( ) shall return the value it has written. Otherwise, it shall
14463 CX return EOF, the error indicator for the stream shall be set, and *errno* shall be set to indicate the
14464 error.

14465 **ERRORS**

14466 The *fputc*( ) function shall fail if either the *stream* is unbuffered or the *stream*'s buffer needs to be
14467 flushed, and:

14468 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and
14469 the thread would be delayed in the write operation.

14470 CX [EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for
14471 writing.

14472 CX [EFBIG] An attempt was made to write to a file that exceeds the maximum file size.

14473 XSI [EFBIG] An attempt was made to write to a file that exceeds the file size limit of the
14474 process.

14475 CX [EFBIG] The file is a regular file and an attempt was made to write at or beyond the
14476 offset maximum.

14477 CX [EINTR] The write operation was terminated due to the receipt of a signal, and no data
14478 was transferred.

14479 CX [EIO] A physical I/O error has occurred, or the process is a member of a background
14480 process group attempting to write to its controlling terminal, TOSTOP is set,
14481 the process is neither ignoring nor blocking SIGTTOU, and the process group
14482 of the process is orphaned. This error may also be returned under
14483 implementation-defined conditions.

14484 CX [ENOSPC] There was no free space remaining on the device containing the file.

14485 CX [EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by
14486 any process. A SIGPIPE signal shall also be sent to the thread.

The *fputc*( ) function may fail if:

CX [ENOMEM] Insufficient storage space is available.

CX [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 1003.1-200x, **<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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/37 is applied, updating the [EAGAIN] error in the ERRRORS section from "the process would be delayed" to "the thread would be delayed".

NAME

fputs — put a string on a stream

SYNOPSIS

```
#include <stdio.h>

int fputs(const char *restrict s, FILE *restrict 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 volume of IEEE Std 1003.1-200x 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.

CX 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 CX 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.

```
#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",
    minutes_to_event);
...
```

APPLICATION USAGE

The *puts*( ) function appends a <newline> while *fputs*( ) does not.

RATIONALE

None.

14555 **FUTURE DIRECTIONS**

14556 None.

14557 **SEE ALSO**

14558 *fopen*( ), *putc*( ), *puts*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

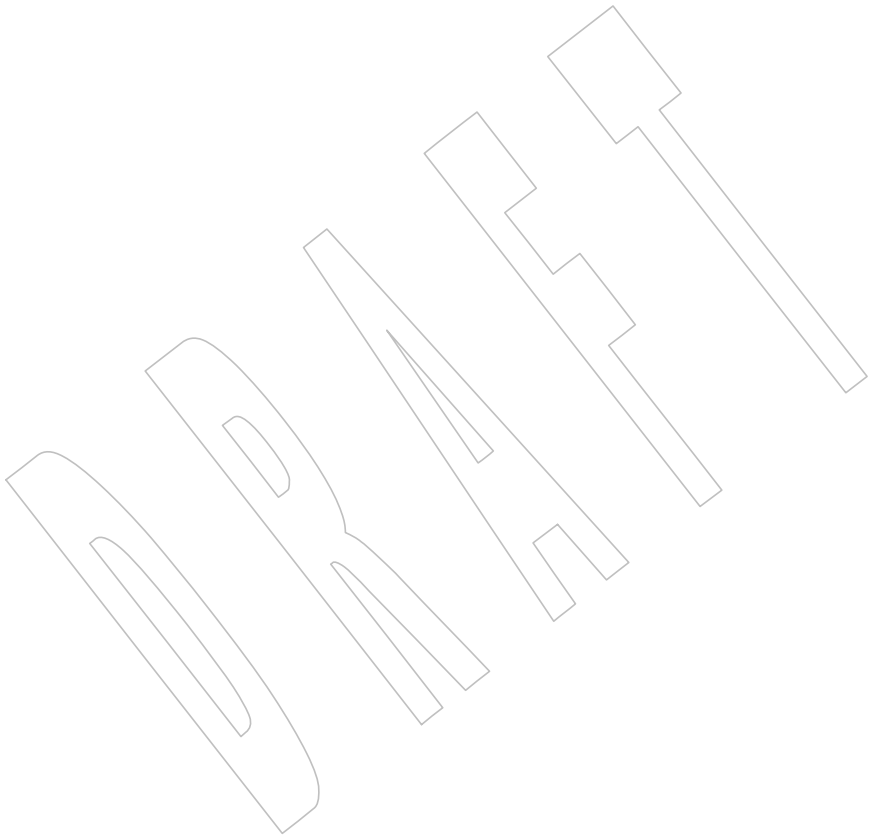
14559 **CHANGE HISTORY**

14560 First released in Issue 1. Derived from Issue 1 of the SVID.

14561 **Issue 6**

14562 Extensions beyond the ISO C standard are marked.

14563 The *fputs*( ) prototype is updated for alignment with the ISO/IEC 9899: 1999 standard.



14564 **NAME**

14565 fputwc — put a wide-character code on a stream

14566 **SYNOPSIS**

14567 ```
#include <stdio.h>
```

14568 ```
#include <wchar.h>
```

14569 ```
wint_t fputwc(wchar_t wc, FILE *stream);
```

14570 **DESCRIPTION**

14571 CX The functionality described on this reference page is aligned with the ISO C standard. Any
14572 conflict between the requirements described here and the ISO C standard is unintentional. This
14573 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

14574 The *fputwc*( ) function shall write the character corresponding to the wide-character code *wc* to
14575 the output stream pointed to by *stream*, at the position indicated by the associated file-position
14576 indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot
14577 support positioning requests, or if the stream was opened with append mode, the character is
14578 appended to the output stream. If an error occurs while writing the character, the shift state of
14579 the output file is left in an undefined state.

14580 CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful
14581 execution of *fputwc*( ) and the next successful completion of a call to *fflush*( ) or *fclose*( ) on the
14582 same stream or a call to *exit*( ) or *abort*( ).

14583 **RETURN VALUE**

14584 Upon successful completion, *fputwc*( ) shall return *wc*. Otherwise, it shall return WEOF, the error
14585 CX indicator for the stream shall be set, and *errno* shall be set to indicate the error.

14586 **ERRORS**

14587 The *fputwc*( ) function shall fail if either the stream is unbuffered or data in the *stream*'s buffer
14588 needs to be written, and:

14589 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and
14590 the thread would be delayed in the write operation.

14591 CX [EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for
14592 writing.

14593 CX [EFBIG] An attempt was made to write to a file that exceeds the maximum file size or
14594 the file size limit of the process.

14595 CX [EFBIG] The file is a regular file and an attempt was made to write at or beyond the
14596 offset maximum associated with the corresponding stream.

14597 [EILSEQ] The wide-character code *wc* does not correspond to a valid character.

14598 CX [EINTR] The write operation was terminated due to the receipt of a signal, and no data
14599 was transferred.

14600 CX [EIO] A physical I/O error has occurred, or the process is a member of a background
14601 process group attempting to write to its controlling terminal, TOSTOP is set,
14602 the process is neither ignoring nor blocking SIGTTOU, and the process group
14603 of the process is orphaned. This error may also be returned under
14604 implementation-defined conditions.

14605 CX [ENOSPC] There was no free space remaining on the device containing the file.

 
```

CX [EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A SIGPIPE signal shall also be sent to the thread.

The *fputwc*( ) function may fail if:

CX [ENOMEM] Insufficient storage space is available.

CX [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 1003.1-200x, **<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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/38 is applied, updating the [EAGAIN] error in the ERRORS section from "the process would be delayed" to "the thread would be delayed".

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

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 1003.1-200x defers to the ISO C standard.

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.

CX 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, CX and set *errno* to indicate the error.

ERRORS

Refer to *fputwc*( ).

EXAMPLES

None.

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 1003.1-200x, **<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

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 1003.1-200x defers to the ISO C standard.

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.

CX The *fread*( ) function may mark the *st_atime* field of the file associated with *stream* for update. The *st_atime* field shall be marked for update by the first successful execution of *fgetc*( ), *fgets*( ), *fgetwc*( ), *fgetws*( ), *fread*( ), *fscanf*( ), *getc*( ), *getchar*( ), *gets*( ), or *scanf*( ) using *stream* that returns data not supplied by a prior call to *ungetc*( ) or *ungetwc*( ).

RETURN VALUE

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.
CX Otherwise, if a read error occurs, the error indicator for the stream shall be set, and *errno* shall be set to indicate the error.

ERRORS

Refer to *fgetc*( ).

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*( ), *getc*( ), *gets*( ), *scanf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

14738 **NAME**

14739 free — free allocated memory

14740 **SYNOPSIS**

14741
```
#include <stdlib.h>
```

14742
```
void free(void *ptr);
```

14743 **DESCRIPTION**

14744 CX The functionality described on this reference page is aligned with the ISO C standard. Any
14745 conflict between the requirements described here and the ISO C standard is unintentional. This
14746 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

14747 The *free*( ) function shall cause the space pointed to by *ptr* to be deallocated; that is, made
14748 available for further allocation. If *ptr* is a null pointer, no action shall occur. Otherwise, if the
14749 ADV argument does not match a pointer earlier returned by the *calloc*( ), *malloc*( ), *posix_memalign*( ),
14750 *realloc*( ), or *strdup*( ) function, or if the space has been deallocated by a call to *free*( ) or *realloc*( ),
14751 the behavior is undefined.

14752 Any use of a pointer that refers to freed space results in undefined behavior.

14753 **RETURN VALUE**

14754 The *free*( ) function shall not return a value.

14755 **ERRORS**

14756 No errors are defined.

14757 **EXAMPLES**

14758 None.

14759 **APPLICATION USAGE**

14760 There is now no requirement for the implementation to support the inclusion of **<malloc.h>**.

14761 **RATIONALE**

14762 None.

14763 **FUTURE DIRECTIONS**

14764 None.

14765 **SEE ALSO**

14766 *calloc*( ), *getdelim*( ), *malloc*( ), *open_memstream*( ), *realloc*( ), *strdup*( ), *wcsdup*( ), the Base Definitions
14767 volume of IEEE Std 1003.1-200x, **<stdlib.h>**

14768 **CHANGE HISTORY**

14769 First released in Issue 1. Derived from Issue 1 of the SVID.

14770 **Issue 6**

14771 Reference to the *valloc*( ) function is removed.

freeaddrinfo( )

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.

Note: In many cases it is implemented by the Domain Name System, as documented in RFC 1034, RFC 1035, and RFC 1886.

The *freeaddrinfo*( ) and *getaddrinfo*( ) f

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, AI_NUMERICSERV, AI_V4MAPPED, AI_ALL, and AI_ADDRCONFIG.

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. The AI_PASSIVE flag shall be ignored if the *nodename* argument is not null.

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).

Note: Since different implementations use different conceptual models, the terms "canonical name" and "alias" cannot be precisely defined for the general case. However, Domain Name System implementations are expected to interpret them as they are used in RFC 1034.

A numeric host address string is not a "name", and thus does not have a "canonical name" form; no address to host name translation is performed. See below for handling of the case where a canonical name cannot be obtained.

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.

IP6 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.

If the AI_ADDRCONFIG flag is specified, IPv4 addresses shall be returned only if an IPv4 IP6 address is configured on the local system, and IPv6 addresses shall be returned only if an IPv6 address is configured on the local system.

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 error 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.

Neither *nodename* nor *servname* were supplied. At least one of these shall be supplied.

[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*.

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.

The term "canonical name" is misleading; it is taken from the Domain Name System (RFC 2181). It should be noted that the canonical name is a result of alias processing, and not necessarily a unique attribute of a host, address, or set of addresses. See RFC 2181 for more discussion of this in the Domain Name System context.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

connect( ), *gai_strerror*( ), *getnameinfo*( ), *getservbyname*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/19 is applied, adding three notes to the DESCRIPTION and adding text to the APPLICATION USAGE related to the term "canonical name". A reference to RFC 2181 is also added to the Informative References.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/20 is applied, making changes for alignment with IPv6. These include the following:

- Adding AI_V4MAPPED, AI_ALL, and AI_ADDRCONFIG to the allowed values for the *ai_flags* field
- Adding a description of AI_ADDRCONFIG
- Adding a description of the consequences of ignoring the AI_PASSIVE flag.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/39 is applied, changing "corresponding value" to "corresponding error value" in the ERRORS section.

Issue 7

Austin Group Interpretation 1003.1-2001 #013 is applied.

14942 **NAME**

14943 freelocale — free resources allocated for a locale object

14944 **SYNOPSIS**

14945 CX
```
#include <locale.h>

void freelocale(locale_t locobj);
```

14947 **DESCRIPTION**

14948 The *freelocale*( ) function shall cause the resources allocated for a locale object returned by a call
14949 to the *newlocale*( ) or *duplocale*( ) functions to be released.

14950 Any use of a locale object that has been freed results in undefined behavior.

14951 **RETURN VALUE**

14952 None.

14953 **ERRORS**

14954 None.

14955 **EXAMPLES**

14956 **Freeing Up a Locale Object**

14957 The following example shows a code fragment to free a locale object created by *newlocale*( ):

```
14958 #include <locale.h>
14959 ...
14960 /* Every locale object allocated with newlocale() should be
14961  * freed using freelocale():
14962  */
14963 locale_t loc;
14964 /* Get the locale. */
14965 loc = newlocale (LC_CTYPE_MASK | LC_TIME_MASK, "locname", NULL);
14966 /* ... Use the locale object ... */
14967 ...
14968 /* Free the locale object resources. */
14969 freelocale (loc);
```

14970 **APPLICATION USAGE**

14971 None.

14972 **RATIONALE**

14973 None.

14974 **FUTURE DIRECTIONS**

14975 None.

14976 **SEE ALSO**

14977 *duplocale*( ), *newlocale*( ), *uselocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
14978 **<locale.h>**

14979 **CHANGE HISTORY**
14980 First released in Issue 7.

NAME

freopen — open a stream

SYNOPSIS

```
#include <stdio.h>

FILE *freopen(const char *restrict filename, const char *restrict mode,
    FILE *restrict 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 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

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*( ).

CX The *freopen*( ) function shall allocate a file descriptor in the same way as *open*( ).

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. In this case, the file descriptor associated with the stream need not be closed if the call to *freopen*( ) succeeds. It is implementation-defined which changes of mode are permitted (if any), and under what circumstances.

XSI 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.

CX 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, CX and *errno* shall be set to indicate the error.

ERRORS

The *freopen*( ) function shall fail if:

CX [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.

CX [EBADF] The file descriptor underlying the stream is not a valid file descriptor when *filename* is a null pointer.

CX [EINTR] A signal was caught during *freopen*( ).

CX [EISDIR] The named file is a directory and *mode* requires write access.

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

15023 CX [EMFILE] All file descriptors available to the process are currently open.

15024 CX [ENAMETOOLONG]
15025 The length of the *filename* argument exceeds {PATH_MAX} or a pathname
15026 component is longer than {NAME_MAX}.

15027 CX [ENFILE] The maximum allowable number of files is currently open in the system.

15028 CX [ENOENT] A component of *filename* does not name an existing file or *filename* is an empty
15029 string.

15030 CX [ENOSPC] The directory or file system that would contain the new file cannot be
15031 expanded, the file does not exist, and it was to be created.

15032 CX [ENOTDIR] A component of the path prefix is not a directory.

15033 CX [ENXIO] The named file is a character special or block special file, and the device
15034 associated with this special file does not exist.

15035 CX [EOVERFLOW] The named file is a regular file and the size of the file cannot be represented
15036 correctly in an object of type **off_t**.

15037 CX [EROFS] The named file resides on a read-only file system and *mode* requires write
15038 access.

15039 The *freopen*( ) function may fail if:

15040 CX [EBADF] The mode with which the file descriptor underlying the stream was opened
15041 does not support the requested mode when *filename* is a null pointer.

15042 CX [EINVAL] The value of the *mode* argument is not valid.

15043 CX [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
15044 resolution of the *path* argument.

15045 CX [ENAMETOOLONG]
15046 Pathname resolution of a symbolic link produced an intermediate result
15047 whose length exceeds {PATH_MAX}.

15048 CX [ENOMEM] Insufficient storage space is available.

15049 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the
15050 capabilities of the device.

15051 CX [ETXTBSY] The file is a pure procedure (shared text) file that is being executed and *mode*
15052 requires write access.

15053 **EXAMPLES**

15054 **Directing Standard Output to a File**

15055 The following example logs all standard output to the **/tmp/logfile** file.

```
15056 #include <stdio.h>
15057 ...
15058 FILE *fp;
15059 ...
15060 fp = freopen ("/tmp/logfile", "a+", stdout);
15061 ...
```


15062 **APPLICATION USAGE**

15063 The *freopen*( ) function is typically used to attach the preopened *streams* associated with *stdin*,
15064 *stdout*, and *stderr* to other files.

15065 **RATIONALE**

15066 None.

15067 **FUTURE DIRECTIONS**

15068 None.

15069 **SEE ALSO**

15070 *fclose*( ), *fdopen*( ), *fmemopen*( ), *fopen*( ), *mbsinit*( ), *open*( ), *open_memstream*( ), the Base Definitions
15071 volume of IEEE Std 1003.1-200x, **<stdio.h>**

15072 **CHANGE HISTORY**

15073 First released in Issue 1. Derived from Issue 1 of the SVID.

15074 **Issue 5**

15075 The DESCRIPTION is updated to indicate that the orientation of the stream is cleared and the
15076 conversion state of the stream is set to an initial conversion state by a successful call to the
15077 *freopen*( ) function.

15078 Large File Summit extensions are added.

15079 **Issue 6**

15080 Extensions beyond the ISO C standard are marked.

15081 The following new requirements on POSIX implementations derive from alignment with the
15082 Single UNIX Specification:

15083 • In the DESCRIPTION, text is added to indicate setting of the offset maximum in the open
15084 file description. This change is to support large files.

15085 • In the ERRORS section, the [EOVERFLOW] condition is added. This change is to support
15086 large files.

15087 • The [ELOOP] mandatory error condition is added.

15088 • A second [ENAMETOOLONG] is added as an optional error condition.

15089 • The [EINVAL], [ENOMEM], [ENXIO], and [ETXTBSY] optional error conditions are added.

15090 The following changes are made for alignment with the ISO/IEC 9899: 1999 standard:

15091 • The *freopen*( ) prototype is updated.

15092 • The DESCRIPTION is updated.

15093 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
15094 [ELOOP] error condition is added.

15095 The DESCRIPTION is updated regarding failure to close, changing the "file" to "file descriptor".

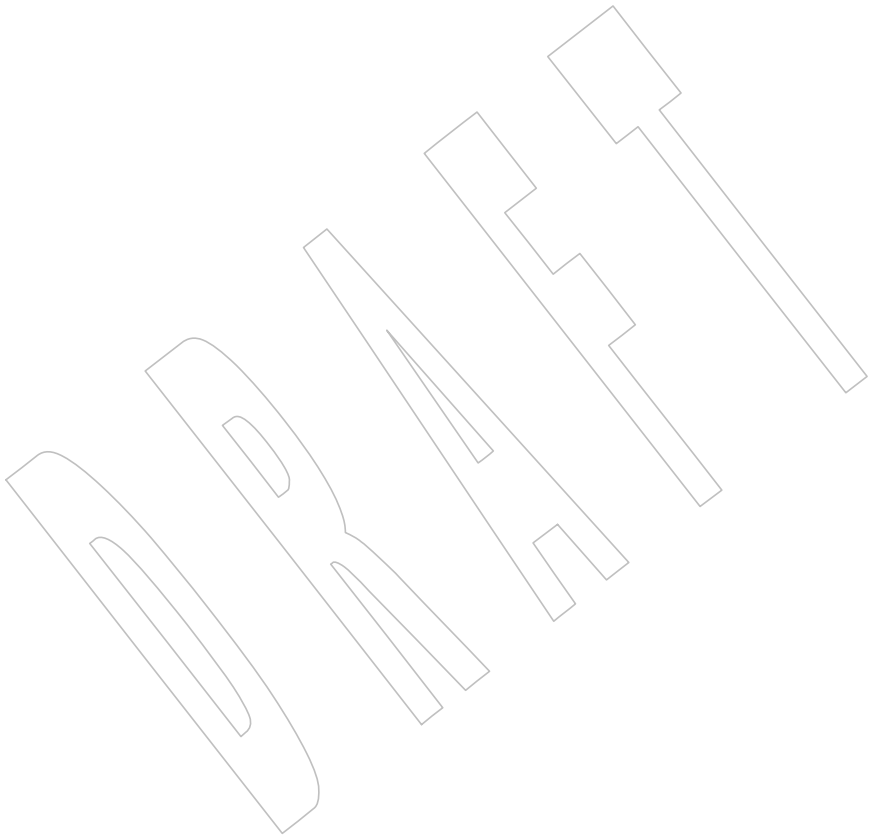
15096 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/40 is applied, adding the following
15097 sentence to the DESCRIPTION: "In this case, the file descriptor associated with the stream need
15098 not be closed if the call to *freopen*( ) succeeds.".

15099 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/41 is applied, adding an mandatory
15100 [EBADF] error, and an optional [EBADF] error to the ERRORS section.

15101 **Issue 7**

15102 SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

15103 Austin Group Interpretation 1003.1-2001 #043 is applied, clarifying that the *freopen*( ) function
15104 allocates a file descriptor as per *open*( ).



NAME

frexp, frexpf, frexpl — extract mantissa and exponent from a double precision number

SYNOPSIS

```
#include <math.h>

double frexp(double num, int *exp);
float frexpf(float num, int *exp);
long double frexpl(long double num, int *exp);
```

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 1003.1-200x 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 [½,1) or 0, and *num* equals *x* times 2 raised to the power **exp*.

MX 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 1003.1-200x, **<math.h>**

CHANGE HISTORY

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

Issue 5

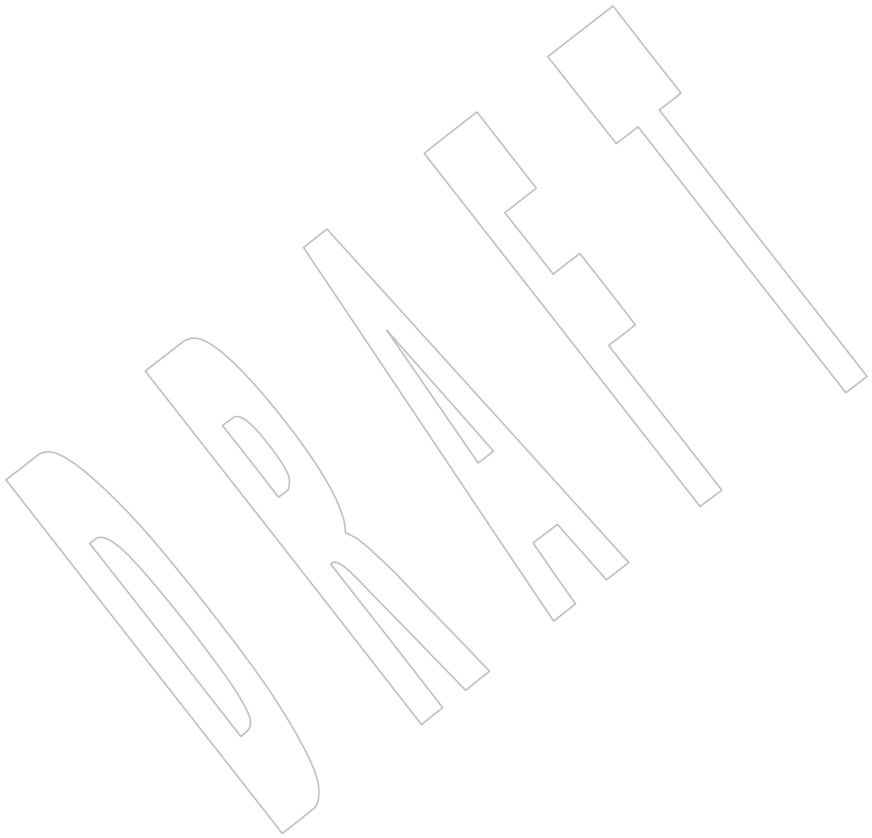
The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

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.

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



15148 **NAME**

15149 fscanf, scanf, sscanf — convert formatted input

15150 **SYNOPSIS**

15151
```
#include <stdio.h>
```

15152
```
int fscanf(FILE *restrict stream, const char *restrict format, ... );
```
15153
```
int scanf(const char *restrict format, ... );
```
15154
```
int sscanf(const char *restrict s, const char *restrict format, ... );
```

15155 **DESCRIPTION**

15156 CX The functionality described on this reference page is aligned with the ISO C standard. Any
15157 conflict between the requirements described here and the ISO C standard is unintentional. This
15158 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

15159 The *fscanf*( ) function shall read from the named input *stream*. The *scanf*( ) function shall read
15160 from the standard input stream *stdin*. The *sscanf*( ) function shall read from the string *s*. Each
15161 function reads bytes, interprets them according to a format, and stores the results in its
15162 arguments. Each expects, as arguments, a control string *format* described below, and a set of
15163 *pointer* arguments indicating where the converted input should be stored. The result is
15164 undefined if there are insufficient arguments for the format. If the format is exhausted while
15165 arguments remain, the excess arguments shall be evaluated but otherwise ignored.

15166 CX Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than
15167 to the next unused argument. In this case, the conversion specifier character % (see below) is
15168 replaced by the sequence "%*n*$", where *n* is a decimal integer in the range [1,{NL_ARGMAX}].
15169 This feature provides for the definition of format strings that select arguments in an order
15170 appropriate to specific languages. In format strings containing the "%*n*$" form of conversion
15171 specifications, it is unspecified whether numbered arguments in the argument list can be
15172 referenced from the format string more than once.

15173 The *format* can contain either form of a conversion specification—that is, % or "%*n*$"—but the
15174 two forms cannot be mixed within a single *format* string. The only exception to this is that %% or
15175 %* can be mixed with the "%*n*$" form. When numbered argument specifications are used,
15176 specifying the *N*th argument requires that all the leading arguments, from the first to the
15177 (*N*–1)th, are pointers.

15178 The *fscanf*( ) function in all its forms shall allow detection of a language-dependent radix
15179 character in the input string. The radix character is defined in the locale of the process (category
15180 *LC_NUMERIC*). In the POSIX locale, or in a locale where the radix character is not defined, the
15181 radix character shall default to a period (`'.'`).

15182 The format is a character string, beginning and ending in its initial shift state, if any, composed
15183 of zero or more directives. Each directive is composed of one of the following: one or more
15184 white-space characters (<space>s, <tab>s, <newline>s, <vertical-tab>s, or <form-feed>s); an
15185 ordinary character (neither `'%'` nor a white-space character); or a conversion specification. Each
15186 CX conversion specification is introduced by the character `'%'` or the character sequence "%*n*$",
15187 after which the following appear in sequence:

15188 - An optional assignment-suppressing character `'*'`.
15189 - An optional non-zero decimal integer that specifies the maximum field width.
15190 - An option length modifier that specifies the size of the receiving object.
15191 - A *conversion specifier* character that specifies the type of conversion to be applied. The valid
15192 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 `'*'`, 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
CX the conversion specification is introduced by `%`, or in the *n*th 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**.

15239 `j` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an
15240 argument with type pointer to **intmax_t** or **uintmax_t**.

15241 `z` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an
15242 argument with type pointer to **size_t** or the corresponding signed integer type.

15243 `t` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an
15244 argument with type pointer to **ptrdiff_t** or the corresponding **unsigned** type.

15245 `L` Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to an
15246 argument with type pointer to **long double**.

15247 If a length modifier appears with any conversion specifier other than as specified above, the
15248 behavior is undefined.

15249 The following conversion specifiers are valid:

15250 `d` Matches an optionally signed decimal integer, whose format is the same as expected for
15251 the subject sequence of *strtol*( ) with the value 10 for the *base* argument. In the absence
15252 of a size modifier, the application shall ensure that the corresponding argument is a
15253 pointer to **int**.

15254 `i` Matches an optionally signed integer, whose format is the same as expected for the
15255 subject sequence of *strtol*( ) with 0 for the *base* argument. In the absence of a size
15256 modifier, the application shall ensure that the corresponding argument is a pointer to
15257 **int**.

15258 `o` Matches an optionally signed octal integer, whose format is the same as expected for
15259 the subject sequence of *strtoul*( ) with the value 8 for the *base* argument. In the absence
15260 of a size modifier, the application shall ensure that the corresponding argument is a
15261 pointer to **unsigned**.

15262 `u` Matches an optionally signed decimal integer, whose format is the same as expected for
15263 the subject sequence of *strtoul*( ) with the value 10 for the *base* argument. In the absence
15264 of a size modifier, the application shall ensure that the corresponding argument is a
15265 pointer to **unsigned**.

15266 `x` Matches an optionally signed hexadecimal integer, whose format is the same as
15267 expected for the subject sequence of *strtoul*( ) with the value 16 for the *base* argument. In
15268 the absence of a size modifier, the application shall ensure that the corresponding
15269 argument is a pointer to **unsigned**.

15270 `a`, `e`, `f`, `g`
15271 Matches an optionally signed floating-point number, infinity, or NaN, whose format is
15272 the same as expected for the subject sequence of *strtod*( ). In the absence of a size
15273 modifier, the application shall ensure that the corresponding argument is a pointer to
15274 **float**.

15275 If the *fprintf*( ) family of functions generates character string representations for infinity
15276 and NaN (a symbolic entity encoded in floating-point format) to support
15277 IEEE Std 754-1985, the *fscanf*( ) family of functions shall recognize them as input.

15278 `s` Matches a sequence of bytes that are not white-space characters. The application shall
15279 ensure that the corresponding argument is a pointer to the initial byte of an array of
15280 **char**, **signed char**, or **unsigned char** large enough to accept the sequence and a
15281 terminating null character code, which shall be added automatically.

15282 If an `l` (ell) qualifier is present, the input is a sequence of characters that begins in the
15283 initial shift state. Each character shall be converted to a wide character as if by a call to
15284 the *mbrtowc*( ) function, with the conversion state described by an **mbstate_t** object
15285 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 *mbrtowc*( ) 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 (`'^'`), 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 `'-'` is in the scanlist and is not the first character, nor the second where the first character is a `'^'`, nor the last character, the behavior is implementation-defined.

c 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 *mbrtowc*( ) 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.

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 specification is undefined.

n 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.

XSI `C` Equivalent to `lc`.

XSI `S` 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.

CX 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 *scanf*( ) 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 CX 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:

CX [EILSEQ] Input byte sequence does not form a valid character.

CX [EINVAL] There are insufficient arguments.

EXAMPLES

The call:

```
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 *name* contains the string `"Hamster"`.

The call:

```
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 *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.

```
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*( ), *strtod*( ), *strtol*( ), *strtoul*( ), *wcrtomb*( ), the Base Definitions volume of IEEE Std 1003.1-200x, 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 `[` 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.

Issue 6

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 normative text 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.

Issue 7

XD5-XSH-ERN-9 is applied, correcting *fscanf*( ) to *scanf*( ) in the DESCRIPTION.

Functionality relating to the %n$ form of conversion specification is moved from the XSI option to the Base.

15429 **NAME**

15430 fseek, fseeko — reposition a file-position indicator in a stream

15431 **SYNOPSIS**

15432
```
#include <stdio.h>
```

15433
```
int fseek(FILE *stream, long offset, int whence);
```

15434 CX
```
int fseeko(FILE *stream, off_t offset, int whence);
```

15435 **DESCRIPTION**

15436 CX The functionality described on this reference page is aligned with the ISO C standard. Any
15437 conflict between the requirements described here and the ISO C standard is unintentional. This
15438 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

15439 The *fseek*( ) function shall set the file-position indicator for the stream pointed to by *stream*. If a
15440 read or write error occurs, the error indicator for the stream shall be set and *fseek*( ) fails.

15441 The new position, measured in bytes from the beginning of the file, shall be obtained by adding
15442 *offset* to the position specified by *whence*. The specified point is the beginning of the file for
15443 SEEK_SET, the current value of the file-position indicator for SEEK_CUR, or end-of-file for
15444 SEEK_END.

15445 If the stream is to be used with wide-character input/output functions, the application shall
15446 ensure that *offset* is either 0 or a value returned by an earlier call to *ftell*( ) on the same stream and
15447 *whence* is SEEK_SET.

15448 A successful call to *fseek*( ) shall clear the end-of-file indicator for the stream and undo any effects
15449 of *ungetc*( ) and *ungetwc*( ) on the same stream. After an *fseek*( ) call, the next operation on an
15450 update stream may be either input or output.

15451 CX If the most recent operation, other than *ftell*( ), on a given stream is *fflush*( ), the file offset in the
15452 underlying open file description shall be adjusted to reflect the location specified by *fseek*( ).

15453 The *fseek*( ) function shall allow the file-position indicator to be set beyond the end of existing
15454 data in the file. If data is later written at this point, subsequent reads of data in the gap shall
15455 return bytes with the value 0 until data is actually written into the gap.

15456 The behavior of *fseek*( ) on devices which are incapable of seeking is implementation-defined.
15457 The value of the file offset associated with such a device is undefined.

15458 If the stream is writable and buffered data had not been written to the underlying file, *fseek*( )
15459 shall cause the unwritten data to be written to the file and shall mark the *st_ctime* and *st_mtime*
15460 fields of the file for update.

15461 In a locale with state-dependent encoding, whether *fseek*( ) restores the stream's shift state is
15462 implementation-defined.

15463 The *fseeko*( ) function shall be equivalent to the *fseek*( ) function except that the *offset* argument is
15464 of type **off_t**.

15465 **RETURN VALUE**

15466 CX The *fseek*( ) and *fseeko*( ) functions shall return 0 if they succeed.

15467 CX Otherwise, they shall return –1 and set *errno* to indicate the error.

15468 **ERRORS**

15469 CX The *fseek*( ) and *fseeko*( ) functions shall fail if, either the *stream* is unbuffered or the *stream*'s
15470 buffer needed to be flushed, and the call to *fseek*( ) or *fseeko*( ) causes an underlying *lseek*( ) or
15471 *write*( ) to be invoked, and:

15472 CX **[EAGAIN]** The O_NONBLOCK flag is set for the file descriptor and the thread would be
15473 delayed in the write operation.

15474 CX **[EBADF]** The file descriptor underlying the stream file is not open for writing or the
15475 stream's buffer needed to be flushed and the file is not open.

15476 CX **[EFBIG]** An attempt was made to write a file that exceeds the maximum file size.

15477 XSI **[EFBIG]** An attempt was made to write a file that exceeds the file size limit of the
15478 process.

15479 CX **[EFBIG]** The file is a regular file and an attempt was made to write at or beyond the
15480 offset maximum associated with the corresponding stream.

15481 CX **[EINTR]** The write operation was terminated due to the receipt of a signal, and no data
15482 was transferred.

15483 CX **[EINVAL]** The *whence* argument is invalid. The resulting file-position indicator would be
15484 set to a negative value.

15485 CX **[EIO]** A physical I/O error has occurred, or the process is a member of a background
15486 process group attempting to perform a *write*( ) to its controlling terminal,
15487 TOSTOP is set, the process is neither ignoring nor blocking SIGTTOU, and the
15488 process group of the process is orphaned. This error may also be returned
15489 under implementation-defined conditions.

15490 CX **[ENOSPC]** There was no free space remaining on the device containing the file.

15491 CX **[ENXIO]** A request was made of a nonexistent device, or the request was outside the
15492 capabilities of the device.

15493 CX **[EOVERFLOW]** For *fseek*( ), the resulting file offset would be a value which cannot be
15494 represented correctly in an object of type **long**.

15495 CX **[EOVERFLOW]** For *fseeko*( ), the resulting file offset would be a value which cannot be
15496 represented correctly in an object of type **off_t**.

15497 CX **[EPIPE]** An attempt was made to write to a pipe or FIFO that is not open for reading
15498 by any process; a SIGPIPE signal shall also be sent to the thread.

15499 CX **[ESPIPE]** The file descriptor underlying *stream* is associated with a pipe or FIFO.

15500 **EXAMPLES**

15501 None.

15502 **APPLICATION USAGE**

15503 None.

15504 **RATIONALE**

15505 None.

15506 **FUTURE DIRECTIONS**

15507 None.

15508 **SEE ALSO**

15509 *fopen*( ), *fsetpos*( ), *ftell*( ), *getrlimit*( ), *lseek*( ), *rewind*( ), *ulimit*( ), *ungetc*( ), *write*( ), the Base
15510 Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

15511 **CHANGE HISTORY**

15512 First released in Issue 1. Derived from Issue 1 of the SVID.

15513 **Issue 5**

15514 Normative text previously in the APPLICATION USAGE section is moved to the
15515 DESCRIPTION.

15516 Large File Summit extensions are added.

15517 **Issue 6**

15518 Extensions beyond the ISO C standard are marked.

15519 The following new requirements on POSIX implementations derive from alignment with the
15520 Single UNIX Specification:

15521 • The *fseeko*( ) function is added.

15522 • The [EFBIG], [EOVERFLOW], and [ENXIO] mandatory error conditions are added.

15523 The following change is incorporated for alignment with the FIPS requirements:

15524 • The [EINTR] error is no longer an indication that the implementation does not report
15525 partial transfers.

15526 The normative text is updated to avoid use of the term "must" for application requirements.

15527 The DESCRIPTION is updated to explicitly state that *fseek*( ) sets the file-position indicator, and
15528 then on error the error indicator is set and *fseek*( ) fails. This is for alignment with the
15529 ISO/IEC 9899:1999 standard.

15530 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/42 is applied, updating the [EAGAIN]
15531 error in the ERRORS section from "the process would be delayed" to "the thread would be
15532 delayed".

15533 **NAME**

15534 fsetpos — set current file position

15535 **SYNOPSIS**

15536
```
#include <stdio.h>
```

15537
```
int fsetpos(FILE *stream, const fpos_t *pos);
```

15538 **DESCRIPTION**

15539 CX The functionality described on this reference page is aligned with the ISO C standard. Any
15540 conflict between the requirements described here and the ISO C standard is unintentional. This
15541 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

15542 The *fsetpos*( ) function shall set the file position and state indicators for the stream pointed to by
15543 *stream* according to the value of the object pointed to by *pos*, which the application shall ensure is
15544 a value obtained from an earlier call to *fgetpos*( ) on the same stream. If a read or write error
15545 occurs, the error indicator for the stream shall be set and *fsetpos*( ) fails.

15546 A successful call to the *fsetpos*( ) function shall clear the end-of-file indicator for the stream and
15547 undo any effects of *ungetc*( ) on the same stream. After an *fsetpos*( ) call, the next operation on an
15548 update stream may be either input or output.

15549 CX The behavior of *fsetpos*( ) on devices which are incapable of seeking is implementation-defined.
15550 The value of the file offset associated with such a device is undefined.

15551 **RETURN VALUE**

15552 The *fsetpos*( ) function shall return 0 if it succeeds; otherwise, it shall return a non-zero value and
15553 set *errno* to indicate the error.

15554 **ERRORS**

15555 CX The *fsetpos*( ) function shall fail if, either the *stream* is unbuffered or the *stream*'s buffer needed to
15556 be flushed, and the call to *fsetpos*( ) causes an underlying *lseek*( ) or *write*( ) to be invoked, and:

15557 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor and the thread would be
15558 delayed in the write operation.

15559 CX [EBADF] The file descriptor underlying the stream file is not open for writing or the
15560 stream's buffer needed to be flushed and the file is not open.

15561 CX [EFBIG] An attempt was made to write a file that exceeds the maximum file size.

15562 XSI [EFBIG] An attempt was made to write a file that exceeds the file size limit of the
15563 process.

15564 CX [EFBIG] The file is a regular file and an attempt was made to write at or beyond the
15565 offset maximum associated with the corresponding stream.

15566 CX [EINTR] The write operation was terminated due to the receipt of a signal, and no data
15567 was transferred.

15568 CX [EIO] A physical I/O error has occurred, or the process is a member of a background
15569 process group attempting to perform a *write*( ) to its controlling terminal,
15570 TOSTOP is set, the process is neither ignoring nor blocking SIGTTOU, and the
15571 process group of the process is orphaned. This error may also be returned
15572 under implementation-defined conditions.

15573 CX [ENOSPC] There was no free space remaining on the device containing the file.

15574 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the
15575 capabilities of the device.

15576 CX [EPIPE] The file descriptor underlying *stream* is associated with a pipe or FIFO.

15577 CX [EPIPE] An attempt was made to write to a pipe or FIFO that is not open for reading
15578 by any process; a SIGPIPE signal shall also be sent to the thread.

15579 **EXAMPLES**

15580 None.

15581 **APPLICATION USAGE**

15582 None.

15583 **RATIONALE**

15584 None.

15585 **FUTURE DIRECTIONS**

15586 None.

15587 **SEE ALSO**

15588 *fopen( )*, *ftell( )*, *lseek( )*, *rewind( )*, *ungetc( )*, *write( )*, the Base Definitions volume of
15589 IEEE Std 1003.1-200x, **<stdio.h>**

15590 **CHANGE HISTORY**

15591 First released in Issue 4. Derived from the ISO C standard.

15592 **Issue 6**

15593 Extensions beyond the ISO C standard are marked.

15594 An additional [ESPIPE] error condition is added for sockets.

15595 The normative text is updated to avoid use of the term "must" for application requirements.

15596 The DESCRIPTION is updated to clarify that the error indicator is set for the stream on a read or
15597 write error. This is for alignment with the ISO/IEC 9899: 1999 standard.

15598 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/21 is applied, deleting an erroneous
15599 [EINVAL] error case from the ERRORS section.

15600 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/43 is applied, updating the [EAGAIN]
15601 error in the ERRORS section from "the process would be delayed" to "the thread would be
15602 delayed".

15603 **NAME**

15604 fstat — get file status

15605 **SYNOPSIS**

15606
```
#include <sys/stat.h>
```

15607
```
int fstat(int fildes, struct stat *buf);
```

15608 **DESCRIPTION**

15609 The *fstat*( ) function shall obtain information about an open file associated with the file
15610 descriptor *fildes*, and shall write it to the area pointed to by *buf*.

15611 SHM If *fildes* references a shared memory object, the implementation shall update in the **stat** structure
15612 pointed to by the *buf* argument the *st_uid*, *st_gid*, *st_size*, and *st_mode* fields, and only the
15613 S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits need be
15614 valid. The implementation may update other fields and flags.

15615 TYM If *fildes* references a typed memory object, the implementation shall update in the **stat** structure
15616 pointed to by the *buf* argument the *st_uid*, *st_gid*, *st_size*, and *st_mode* fields, and only the
15617 S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits need be
15618 valid. The implementation may update other fields and flags.

15619 The *buf* argument is a pointer to a **stat** structure, as defined in **<sys/stat.h>**, into which
15620 information is placed concerning the file.

15621 For all other file types defined in this volume of IEEE Std 1003.1-200x, the structure members
15622 *st_mode*, *st_ino*, *st_dev*, *st_uid*, *st_gid*, *st_atime*, *st_ctime*, and *st_mtime* shall have meaningful
15623 values and the value of the *st_nlink* member shall be set to the number of links to the file.

15624 An implementation that provides additional or alternative file access control mechanisms may,
15625 under implementation-defined conditions, cause *fstat*( ) to fail.

15626 The *fstat*( ) function shall update any time-related fields as described in the Base Definitions
15627 volume of IEEE Std 1003.1-200x, Section 4.7, File Times Update, before writing into the **stat**
15628 structure.

15629 **RETURN VALUE**

15630 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to
15631 indicate the error.

15632 **ERRORS**

15633 The *fstat*( ) function shall fail if:

15634 [EBADF] The *fildes* argument is not a valid file descriptor.

15635 [EIO] An I/O error occurred while reading from the file system.

15636 [EOVERFLOW] The file size in bytes or the number of blocks allocated to the file or the file
15637 serial number cannot be represented correctly in the structure pointed to by
15638 *buf*.

15639 The *fstat*( ) function may fail if:

15640 [EOVERFLOW] One of the values is too large to store into the structure pointed to by the *buf*
15641 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*.

```
#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

fstatat( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

15682 **Issue 7**

15683 XSH-SD5-ERN-161 is applied, updating the DESCRIPTION to clarify which file types *st_nlink*
15684 applies to.

15685 **NAME**

15686 fstatat, lstat, stat — get file status

15687 **SYNOPSIS**

15688
```
#include <sys/stat.h>
```

15689
```
int fstatat(int fd, const char *restrict path,
    struct stat *restrict buf);
```

15690
```
int lstat(const char *restrict path, struct stat *restrict buf);
```

15691
```
int stat(const char *restrict path, struct stat *restrict buf);
```

15692 **DESCRIPTION**

15693 The *stat*( ) function shall obtain information about the named file and write it to the area pointed
15694 to by the *buf* argument. The *path* argument points to a pathname naming a file. Read, write, or
15695 execute permission of the named file is not required. An implementation that provides
15696 additional or alternate file access control mechanisms may, under implementation-defined
15697 conditions, cause *stat*( ) to fail. In particular, the system may deny the existence of the file
15698 specified by *path*.

15699 If the named file is a symbolic link, the *stat*( ) function shall continue pathname resolution using
15700 the contents of the symbolic link, and shall return information pertaining to the resulting file if
15701 the file exists.

15702 The *buf* argument is a pointer to a **stat** structure, as defined in the **<sys/stat.h>** header, into
15703 which information is placed concerning the file.

15704 The *stat*( ) function shall update any time-related fields (as described in the Base Definitions
15705 volume of IEEE Std 1003.1-200x, Section 4.7, File Times Update), before writing into the **stat**
15706 structure.

15707 SHM If the named file is a shared memory object, the implementation shall update in the **stat** structure
15708 pointed to by the *buf* argument the *st_uid*, *st_gid*, *st_size*, and *st_mode* fields, and only the
15709 S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits need be
15710 valid. The implementation may update other fields and flags.

15711 TYM If the named file is a typed memory object, the implementation shall update in the **stat** structure
15712 pointed to by the *buf* argument the *st_uid*, *st_gid*, *st_size*, and *st_mode* fields, and only the
15713 S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits need be
15714 valid. The implementation may update other fields and flags.

15715 For all other file types defined in this volume of IEEE Std 1003.1-200x, the structure members
15716 *st_mode*, *st_ino*, *st_dev*, *st_uid*, *st_gid*, *st_atime*, *st_ctime*, and *st_mtime* shall have meaningful
15717 values and the value of the member *st_nlink* shall be set to the number of links to the file.

15718 The *lstat*( ) function shall be equivalent to *stat*( ), except when *path* refers to a symbolic link. In
15719 that case *lstat*( ) shall return information about the link, while *stat*( ) shall return information
15720 about the file the link references.

15721 For symbolic links, the *st_mode* member shall contain meaningful information when used with
15722 the file type macros, and the *st_size* member shall contain the length of the pathname contained
15723 in the symbolic link. File mode bits and the contents of the remaining members of the **stat**
15724 structure are unspecified. The value returned in the *st_size* member is the length of the contents
15725 of the symbolic link, and does not count any trailing null.

15726 The *fstatat*( ) function shall be equivalent to the *stat*( ) or *lstat*( ) function, depending on the value
15727 of *flag* (see below), except in the case where *path* specifies a relative path. In this case the status
15728 shall be retrieved from a file relative to the directory associated with the file descriptor *fd* instead
15729 of the current working directory. It is unspecified whether directory searches are permitted

based on whether the file was opened with search permission or on the current permissions of the directory underlying the file descriptor.

Values for *flag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in **<fcntl.h>**:

AT_SYMLINK_NOFOLLOW
: If *path* names a symbolic link, the status of the symbolic link is returned.

If *fstatat*( ) is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior shall be identical to a call to *stat*( ) or *lstat*( ) respectively, depending on whether or not the AT_SYMLINK_NOFOLLOW bit is set in *flag*.

RETURN VALUE

Upon successful completion, these functions shall return 0. Otherwise, these functions shall return −1 and set *errno* to indicate the error.

ERRORS

These functions 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 *fstatat*( ) function shall fail if:

[EBADF] The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

These functions 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] A value to be stored would overflow one of the members of the **stat** structure.

The *fstatat*( ) function may fail if:

[EINVAL] The value of the *flag* argument is not valid.

[ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

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.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

struct stat buffer;
int         status;
...
status = stat("/home/cnd/mod1", &buffer);
```

Getting Directory Information

The following example fragment gets status information for each entry in a directory. The call to the *stat*( ) function stores file information in the **stat** structure pointed to by *statbuf*. The lines that follow the *stat*( ) call format the fields in the **stat** structure for presentation to the user of the program.

```
#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) {
```

```
        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\n", datestring, dp->d_name);
    }
```

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.

```
#include <sys/stat.h>

struct stat buffer;
int status;
...
status = lstat("/modules/pass1", &buffer);
```

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.

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 1003.1-200x are required to maintain any of these time fields.

The purpose of the *fstatat*( ) function is to obtain the status of files in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to *stat*( ), resulting in unspecified behavior. By opening a file descriptor for the target directory and using the *fstatat*( ) function it can be guaranteed that the file for which status is returned is located relative to the desired directory.

FUTURE DIRECTIONS

None.

SEE ALSO

access( ), *chmod( )*, *fdopendir( )*, *fstat( )*, *mknod( )*, *readlink( )*, *symlink( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<fcntl.h>**, **<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 normative text 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.

Issue 7

The *fstatat( )* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 2.

XSH-SD5-ERN-161 is applied, updating the DESCRIPTION to clarify which file types *st_nlink* applies to.

15891 **NAME**

15892 fstatvfs, statvfs — get file system information

15893 **SYNOPSIS**

15894
```
#include <sys/statvfs.h>
```

15895
```
int fstatvfs(int fildes, struct statvfs *buf);
```
15896
```
int statvfs(const char *restrict path, struct statvfs *restrict buf);
```

15897 **DESCRIPTION**

15898 The *fstatvfs*( ) function shall obtain information about the file system containing the file
15899 referenced by *fildes*.

15900 The *statvfs*( ) function shall obtain information about the file system containing the file named by
15901 *path*.

15902 For both functions, the *buf* argument is a pointer to a **statvfs** structure that shall be filled. Read,
15903 write, or execute permission of the named file is not required.

15904 The following flags can be returned in the *f_flag* member:

15905 ST_RDONLY Read-only file system.

15906 ST_NOSUID Setuid/setgid bits ignored by *exec*.

15907 It is unspecified whether all members of the **statvfs** structure have meaningful values on all file
15908 systems.

15909 **RETURN VALUE**

15910 Upon successful completion, *statvfs*( ) shall return 0. Otherwise, it shall return –1 and set *errno* to
15911 indicate the error.

15912 **ERRORS**

15913 The *fstatvfs*( ) and *statvfs*( ) functions shall fail if:

15914 [EIO] An I/O error occurred while reading the file system.

15915 [EINTR] A signal was caught during execution of the function.

15916 [EOVERFLOW] One of the values to be returned cannot be represented correctly in the
15917 structure pointed to by *buf*.

15918 The *fstatvfs*( ) function shall fail if:

15919 [EBADF] The *fildes* argument is not an open file descriptor.

15920 The *statvfs*( ) function shall fail if:

15921 [EACCES] Search permission is denied on a component of the path prefix.

15922 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
15923 argument.

15924 [ENAMETOOLONG]
15925 The length of a pathname exceeds {PATH_MAX} or a pathname component is
15926 longer than {NAME_MAX}.

15927 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

15928 [ENOTDIR] A component of the path prefix of *path* is not a directory.

The *statvfs*( ) 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

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.

```
#include <sys/statvfs.h>
#include <fcntl.h>

struct statvfs buffer;
int            status;
...
fildes = open("/home/cnd/mod1", O_RDWR);
status  = fstatvfs(fildes, &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.

```
#include <sys/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 1003.1-200x, **<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 normative text 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.

Issue 7

The *fstatvfs*( ) and *statvfs*( ) functions are moved from the XSI option to the Base.

SD5-XSH-ERN-68 is applied, correcting the EXAMPLES section.

15979 **NAME**

15980 fsync — synchronize changes to a file

15981 **SYNOPSIS**

15982 FSC
```
#include <unistd.h>

int fsync(int fildes);
```
15983

15984 **DESCRIPTION**

15985 The *fsync*( ) function shall request that all data for the open file descriptor named by *fildes* is to be
15986 transferred to the storage device associated with the file described by *fildes*. The nature of the
15987 transfer is implementation-defined. The *fsync*( ) function shall not return until the system has
15988 completed that action or until an error is detected.

15989 SIO If _POSIX_SYNCHRONIZED_IO is defined, the *fsync*( ) function shall force all currently queued
15990 I/O operations associated with the file indicated by file descriptor *fildes* to the synchronized I/O
15991 completion state. All I/O operations shall be completed as defined for synchronized I/O file
15992 integrity completion.

15993 **RETURN VALUE**

15994 Upon successful completion, *fsync*( ) shall return 0. Otherwise, –1 shall be returned and *errno* set
15995 to indicate the error. If the *fsync*( ) function fails, outstanding I/O operations are not guaranteed
15996 to have been completed.

15997 **ERRORS**

15998 The *fsync*( ) function shall fail if:

15999 [EBADF] The *fildes* argument is not a valid descriptor.

16000 [EINTR] The *fsync*( ) function was interrupted by a signal.

16001 [EINVAL] The *fildes* argument does not refer to a file on which this operation is possible.

16002 [EIO] An I/O error occurred while reading from or writing to the file system.

16003 In the event that any of the queued I/O operations fail, *fsync*( ) shall return the error conditions
16004 defined for *read*( ) and *write*( ).

16005 **EXAMPLES**

16006 None.

16007 **APPLICATION USAGE**

16008 The *fsync*( ) function should be used by programs which require modifications to a file to be
16009 completed before continuing; for example, a program which contains a simple transaction
16010 facility might use it to ensure that all modifications to a file or files caused by a transaction are
16011 recorded.

16012 **RATIONALE**

16013 The *fsync*( ) function is intended to force a physical write of data from the buffer cache, and to
16014 assure that after a system crash or other failure that all data up to the time of the *fsync*( ) call is
16015 recorded on the disk. Since the concepts of "buffer cache", "system crash", "physical write", and
16016 "non-volatile storage" are not defined here, the wording has to be more abstract.

16017 If _POSIX_SYNCHRONIZED_IO is not defined, the wording relies heavily on the conformance
16018 document to tell the user what can be expected from the system. It is explicitly intended that a
16019 null implementation is permitted. This could be valid in the case where the system cannot assure
16020 non-volatile storage under any circumstances or when the system is highly fault-tolerant and the
16021 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 *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 *fsync*( ), allowing it to be treated as a quality-of-implementation issue.

FUTURE DIRECTIONS

None.

SEE ALSO

sync( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 3.

Issue 5

Aligned with *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 *fsync*( ) can return the error conditions defined for *read*( ) and *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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/44 is applied, applying an editorial rewording of the DESCRIPTION. No change in meaning is intended.

NAME

ftell, ftello — return a file offset in a stream

SYNOPSIS

```
#include <stdio.h>

long ftell(FILE *stream);
CX off_t ftello(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 volume of IEEE Std 1003.1-200x 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*.

CX The *ftello*( ) function shall be equivalent to *ftell*( ), except that the return value is of type **off_t**.

RETURN VALUE

CX Upon successful completion, *ftell*( ) and *ftello*( ) shall return the current value of the file-position

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.

16101 **NAME**

16102 ftok — generate an IPC key

16103 **SYNOPSIS**

16104 XSI
```
#include <sys/ipc.h>

key_t ftok(const char *path, int id);
```

16105

16106 **DESCRIPTION**

16107 The *ftok*( ) function shall return a key based on *path* and *id* that is usable in subsequent calls to
16108 *msgget*( ), *semget*( ), and *shmget*( ). The application shall ensure that the *path* argument is the
16109 pathname of an existing file that the process is able to *stat*( ).

16110 The *ftok*( ) function shall return the same key value for all paths that name the same file, when
16111 called with the same *id* value, and return different key values when called with different *id*
16112 values or with paths that name different files existing on the same file system at the same time. It
16113 is unspecified whether *ftok*( ) shall return the same key value when called again after the file
16114 named by *path* is removed and recreated with the same name.

16115 Only the low-order 8-bits of *id* are significant. The behavior of *ftok*( ) is unspecified if these bits
16116 are 0.

16117 **RETURN VALUE**

16118 Upon successful completion, *ftok*( ) shall return a key. Otherwise, *ftok*( ) shall return (**key_t**)–1
16119 and set *errno* to indicate the error.

16120 **ERRORS**

16121 The *ftok*( ) function shall fail if:

16122 [EACCES] Search permission is denied for a component of the path prefix.

16123 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
16124 argument.

16125 [ENAMETOOLONG]
16126 The length of the *path* argument exceeds {PATH_MAX} or a pathname
16127 component is longer than {NAME_MAX}.

16128 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

16129 [ENOTDIR] A component of the path prefix is not a directory.

16130 The *ftok*( ) function may fail if:

16131 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
16132 resolution of the *path* argument.

16133 [ENAMETOOLONG]
16134 Pathname resolution of a symbolic link produced an intermediate result
16135 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**.

```
#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*( ).

```
#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 1003.1-200x, **<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 normative text 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.

16173 **NAME**

16174 ftruncate — truncate a file to a specified length

16175 **SYNOPSIS**

16176
```
#include <unistd.h>
```

16177
```
int ftruncate(int fildes, off_t length);
```

16178 **DESCRIPTION**

16179 If *fildes* is not a valid file descriptor open for writing, the *ftruncate*( ) function shall fail.

16180 If *fildes* refers to a regular file, the *ftruncate*( ) function shall cause the size of the file to be
16181 truncated to *length*. If the size of the file previously exceeded *length*, the extra data shall no
16182 longer be available to reads on the file. If the file previously was smaller than this size,
16183 *ftruncate*( ) shall increase the size of the file. If the file size is increased, the extended area shall
16184 appear as if it were zero-filled. The value of the seek pointer shall not be modified by a call to
16185 *ftruncate*( ).

16186 Upon successful completion, if *fildes* refers to a regular file, the *ftruncate*( ) function shall mark
16187 for update the *st_ctime* and *st_mtime* fields of the file and the S_ISUID and S_ISGID bits of the
16188 file mode may be cleared. If the *ftruncate*( ) function is unsuccessful, the file is unaffected.

16189 XSI If the request would cause the file size to exceed the soft file size limit for the process, the
16190 request shall fail and the implementation shall generate the SIGXFSZ signal for the thread.

16191 If *fildes* refers to a directory, *ftruncate*( ) shall fail.

16192 If *fildes* refers to any other file type, except a shared memory object, the result is unspecified.

16193 SHM If *fildes* refers to a shared memory object, *ftruncate*( ) shall set the size of the shared memory
16194 object to *length*.

16195 SHM If the effect of *ftruncate*( ) is to decrease the size of a memory mapped file or a shared memory
16196 object and whole pages beyond the new end were previously mapped, then the whole pages
16197 beyond the new end shall be discarded.

16198 References to discarded pages shall result in the generation of a SIGBUS signal.

16199 If the effect of *ftruncate*( ) is to increase the size of a memory object, it is unspecified whether the
16200 contents of any mapped pages between the old end-of-file and the new are flushed to the
16201 underlying object.

16202 **RETURN VALUE**

16203 Upon successful completion, *ftruncate*( ) shall return 0; otherwise, –1 shall be returned and *errno*
16204 set to indicate the error.

16205 **ERRORS**

16206 The *ftruncate*( ) function shall fail if:

16207 [EINTR] A signal was caught during execution.

16208 [EINVAL] The *length* argument was less than 0.

16209 [EFBIG] or [EINVAL]
16210 The *length* argument was greater than the maximum file size.

16211 [EFBIG] The file is a regular file and *length* is greater than the offset maximum
16212 established in the open file description associated with *fildes*.

[EIO] An I/O error occurred while reading from or writing to a file system.

[EBADF] or [EINVAL]
The *fildes* argument is not a file descriptor open for writing.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

open( ), *truncate*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE and aligned with *ftruncate*( ) in the POSIX Realtime Extension. Specifically, the DESCRIPTION is extensively reworded and [EROFS] is added to the list of mandatory errors that can be returned by *ftruncate*( ).

Large File Summit extensions are added.

Issue 6

The *truncate*( ) function is split out into a separate reference page.

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

- The DESCRIPTION is changed to indicate that if the file size is changed, and if the file is a regular file, the S_ISUID and S_ISGID bits in the file mode may be cleared.

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

- The DESCRIPTION text is updated.

XSI-conformant systems are required to increase the size of the file if the file was previously smaller than the size requested.

Issue 7

Austin Group Interpretation 1003.1-2001 #056 is applied, revising the ERRORS section (although the [EINVAL] "may fail" error was subsequently removed during review of the XSI option).

Functionality relating to the Memory Protection and Memory Mapped Files options is moved to the Base.

The DESCRIPTION is updated so that a call to *ftruncate*( ) when the file is smaller than the size requested will increase the size of the file. Previously, non-XSI-conforming implementations were allowed to increase the size of the file or fail.

16251 **NAME**

16252 ftrylockfile — stdio locking functions

16253 **SYNOPSIS**

16254 CX
```
#include <stdio.h>

int ftrylockfile(FILE *file);
```

16256 **DESCRIPTION**

16257 Refer to *flockfile( )*.

16258 **NAME**

16259 ftw — traverse (walk) a file tree

16260 **SYNOPSIS**

16261 OB XSI
```
#include <ftw.h>

int ftw(const char *path, int (*fn)(const char *,
    const struct stat *ptr, int flag), int ndirs);
```

16264 **DESCRIPTION**

16265 The *ftw*( ) function shall recursively descend the directory hierarchy rooted in *path*. For each
16266 object in the hierarchy, *ftw*( ) shall call the function pointed to by *fn*, passing it a pointer to a null-
16267 terminated character string containing the name of the object, a pointer to a **stat** structure
16268 containing information about the object, filled in as if *stat*( ) or *lstat*( ) had been called to retrieve
16269 the information. Possible values of the integer, defined in the **<ftw.h>** header, are:

16270 FTW_D For a directory.

16271 FTW_DNR For a directory that cannot be read.

16272 FTW_F For a file.

16273 FTW_SL For a symbolic link (but see also FTW_NS below).

16274 FTW_NS For an object other than a symbolic link on which *stat*( ) could not successfully be
16275 executed. If the object is a symbolic link and *stat*( ) failed, it is unspecified whether
16276 *ftw*( ) passes FTW_SL or FTW_NS to the user-supplied function.

16277 If the integer is FTW_DNR, descendants of that directory shall not be processed. If the integer is
16278 FTW_NS, the **stat** structure contains undefined values. An example of an object that would
16279 cause FTW_NS to be passed to the function pointed to by *fn* would be a file in a directory with
16280 read but without execute (search) permission.

16281 The *ftw*( ) function shall visit a directory before visiting any of its descendants.

16282 The *ftw*( ) function shall use at most one file descriptor for each level in the tree.

16283 The argument *ndirs* should be in the range [1,{OPEN_MAX}].

16284 The tree traversal shall continue until either the tree is exhausted, an invocation of *fn* returns a
16285 non-zero value, or some error, other than [EACCES], is detected within *ftw*( ).

16286 The *ndirs* argument shall specify the maximum number of directory streams or file descriptors
16287 or both available for use by *ftw*( ) while traversing the tree. When *ftw*( ) returns it shall close any
16288 directory streams and file descriptors it uses not counting any opened by the application-
16289 supplied *fn* function.

16290 The results are unspecified if the application-supplied *fn* function does not preserve the current
16291 working directory.

16292 The *ftw*( ) function need not be thread-safe. A function that is not required to be thread-safe is
16293 not required to be reentrant.

16294 **RETURN VALUE**

16295 If the tree is exhausted, *ftw*( ) shall return 0. If the function pointed to by *fn* returns a non-zero
16296 value, *ftw*( ) shall stop its tree traversal and return whatever value was returned by the function
16297 pointed to by *fn*. If *ftw*( ) detects an error, it shall return –1 and set *errno* to indicate the error.

16298 If *ftw*( ) encounters an error other than [EACCES] (see FTW_DNR and FTW_NS above), it shall
16299 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 *stat* functions is executed on a directory or file.

ERRORS

The *ftw*( ) function shall fail if:

[EACCES] Search permission is denied for any component of *path* or read permission is denied for *path*.

[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 *ftw*( ) function may fail if:

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

[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}.

In addition, if the function pointed to by *fn* encounters system errors, *errno* may be set accordingly.

EXAMPLES

Walking a Directory Structure

The following example walks the current directory structure, calling the *fn* function for every directory entry, using at most 10 file descriptors:

```
#include <ftw.h>
...
if (ftw(".", fn, 10) != 0) {
    perror("ftw"); exit(2);
}
```

APPLICATION USAGE

The *ftw*( ) function may allocate dynamic storage during its operation. If *ftw*( ) is forcibly terminated, such as by *longjmp*( ) or *siglongjmp*( ) being executed by the function pointed to by *fn* or an interrupt routine, *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 *fn* return a non-zero value at its next invocation.

Applications should use the *nftw*( ) function instead of the obsolescent *ftw*( ) function.

16341 **RATIONALE**

16342 None.

16343 **FUTURE DIRECTIONS**

16344 The *ftw*( ) function may be removed in a future version.

16345 **SEE ALSO**

16346 *fdopendir*( ), *fstatat*( ), *longjmp*( ), *malloc*( ), *nftw*( ), *siglongjmp*( ), the Base Definitions volume of
16347 IEEE Std 1003.1-200x, **<ftw.h>**, **<sys/stat.h>**

16348 **CHANGE HISTORY**

16349 First released in Issue 1. Derived from Issue 1 of the SVID.

16350 **Issue 5**

16351 UX codings in the DESCRIPTION, RETURN VALUE, and ERRORS sections are changed to EX.

16352 **Issue 6**

16353 The ERRORS section is updated as follows:

16354 • The wording of the mandatory [ELOOP] error condition is updated.

16355 • A second optional [ELOOP] error condition is added.

16356 • The [EOVERFLOW] mandatory error condition is added.

16357 A note is added to the DESCRIPTION indicating that this function need not be reentrant, and
16358 that the results are unspecified if the application-supplied *fn* function does not preserve the
16359 current working directory.

16360 **Issue 7**

16361 SD5-XBD-ERN-61 is applied.

16362 The *ftw*( ) function ia marked obsolescent.

16363 **NAME**

16364 funlockfile — stdio locking functions

16365 **SYNOPSIS**

16366 CX
```
#include <stdio.h>

void funlockfile(FILE *file);
```

16368 **DESCRIPTION**

16369 Refer to *flockfile( )*.

16370 **NAME**

16371 futimesat — set file access and modification times relative to directory file descriptor

16372 **SYNOPSIS**

16373 XSI
```
#include <sys/time.h>

int futimesat(int fd, const char *path, const struct timeval times[2]);
```

16375 **DESCRIPTION**

16376 Refer to *utimes( )*.

16377 **NAME**

16378 fwide — set stream orientation

16379 **SYNOPSIS**

16380 ```
#include <stdio.h>
```

16381 ```
#include <wchar.h>
```

16382 ```
int fwide(FILE *stream, int mode);
```

16383 **DESCRIPTION**

16384 CX The functionality described on this reference page is aligned with the ISO C standard. Any
16385 conflict between the requirements described here and the ISO C standard is unintentional. This
16386 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

16387 The *fwide*( ) function shall determine the orientation of the stream pointed to by *stream*. If *mode* is
16388 greater than zero, the function first attempts to make the stream wide-oriented. If *mode* is less
16389 than zero, the function first attempts to make the stream byte-oriented. Otherwise, *mode* is zero
16390 and the function does not alter the orientation of the stream.

16391 If the orientation of the stream has already been determined, *fwide*( ) shall not change it.

16392 CX Since no return value is reserved to indicate an error, an application wishing to check for error
16393 situations should set *errno* to 0, then call *fwide*( ), then check *errno*, and if it is non-zero, assume
16394 an error has occurred.

16395 **RETURN VALUE**

16396 The *fwide*( ) function shall return a value greater than zero if, after the call, the stream has wide-
16397 orientation, a value less than zero if the stream has byte-orientation, or zero if the stream has no
16398 orientation.

16399 **ERRORS**

16400 The *fwide*( ) function may fail if:

16401 CX [EBADF] The *stream* argument is not a valid stream.

16402 **EXAMPLES**

16403 None.

16404 **APPLICATION USAGE**

16405 A call to *fwide*( ) with *mode* set to zero can be used to determine the current orientation of a
16406 stream.

16407 **RATIONALE**

16408 None.

16409 **FUTURE DIRECTIONS**

16410 None.

16411 **SEE ALSO**

16412 The Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

16413 **CHANGE HISTORY**

16414 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
16415 (E).

16416 **Issue 6**

16417 Extensions beyond the ISO C standard are marked.

  
```

16418 **NAME**

16419 fwprintf, swprintf, wprintf — print formatted wide-character output

16420 **SYNOPSIS**

16421 ```
#include <stdio.h>
```
16422 ```
#include <wchar.h>
```

16423 ```
int fwprintf(FILE *restrict stream, const wchar_t *restrict format, ...);
```
16424 ```
int swprintf(wchar_t *restrict ws, size_t n,
```
16425 ```
    const wchar_t *restrict format, ...);
```
16426 ```
int wprintf(const wchar_t *restrict format, ...);
```

16427 **DESCRIPTION**

16428 CX The functionality described on this reference page is aligned with the ISO C standard. Any
16429 conflict between the requirements described here and the ISO C standard is unintentional. This
16430 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

16431 The *fwprintf*( ) function shall place output on the named output *stream*. The *wprintf*( ) function
16432 shall place output on the standard output stream *stdout*. The *swprintf*( ) function shall place
16433 output followed by the null wide character in consecutive wide characters starting at **ws*; no
16434 more than *n* wide characters shall be written, including a terminating null wide character, which
16435 is always added (unless *n* is zero).

16436 Each of these functions shall convert, format, and print its arguments under control of the *format*
16437 wide-character string. The *format* is composed of zero or more directives: *ordinary wide-characters*,
16438 which are simply copied to the output stream, and *conversion specifications*, each of which results
16439 in the fetching of zero or more arguments. The results are undefined if there are insufficient
16440 arguments for the *format*. If the *format* is exhausted while arguments remain, the excess
16441 arguments are evaluated but are otherwise ignored.

16442 CX Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than
16443 to the next unused argument. In this case, the conversion specifier wide character % (see below)
16444 is replaced by the sequence "%*n*$", where *n* is a decimal integer in the range
16445 [1,{NL_ARGMAX}], giving the position of the argument in the argument list. This feature
16446 provides for the definition of *format* wide-character strings that select arguments in an order
16447 appropriate to specific languages (see the EXAMPLES section).

16448 The *format* can contain either numbered argument specifications (that is, "%*n*$" and "**m*$"), or
16449 unnumbered argument conversion specifications (that is, % and *), but not both. The only
16450 exception to this is that %% can be mixed with the "%*n*$" form. The results of mixing numbered
16451 and unnumbered argument specifications in a *format* wide-character string are undefined. When
16452 numbered argument specifications are used, specifying the *N*th argument requires that all the
16453 leading arguments, from the first to the (*N*–1)th, are specified in the *format* wide-character string.

16454 In *format* wide-character strings containing the "%*n*$" form of conversion specification,
16455 numbered arguments in the argument list can be referenced from the *format* wide-character
16456 string as many times as required.

16457 In *format* wide-character strings containing the % form of conversion specification, each
16458 argument in the argument list shall be used exactly once.

16459 CX All forms of the *fwprintf*( ) function allow for the insertion of a locale-dependent radix character
16460 in the output string, output as a wide-character value. The radix character is defined in the
16461 locale of the process (category *LC_NUMERIC*). In the POSIX locale, or in a locale where the
16462 radix character is not defined, the radix character shall default to a period ('.').

16463 CX Each conversion specification is introduced by the '%' wide character or by the wide-character
16464 sequence "%*n*$", after which the following appear in sequence:

16465 16466 • Zero or more *flags* (in any order), which modify the meaning of the conversion specification.

16467 16468 16469 16470 • 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.

16471 16472 16473 16474 16475 16476 16477 16478 • 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.

16479 • An optional length modifier that specifies the size of the argument.

16480 • A *conversion specifier* wide character that indicates the type of conversion to be applied.

16481 16482 16483 16484 16485 CX 16486 16487 16488 16489 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:

16490
```
wprintf(L"%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```

16491 The flag wide characters and their meanings are:

16492 CX ' The integer portion of the result of a decimal conversion (%i, %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.

16495 16496 – The result of the conversion shall be left-justified within the field. The conversion shall be right-justified if this flag is not specified.

16497 16498 16499 + 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.

16500 16501 16502 <space> 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.

16503 16504 16505 16506 16507 16508 16509 16510 # 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.

`0` For `d`, `i`, `o`, `u`, `x`, `X`, `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversion specifiers, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the `'0'` and `'-'` flags both appear, the `'0'` flag shall be ignored. For `d`, `i`, `o`, `u`, `x`, and `X` conversion specifiers, if a precision is specified, the `'0'` flag shall be ignored. CX 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.

16557 o The **unsigned** argument shall be converted to unsigned octal format in the style
16558 "dddd". The precision specifies the minimum number of digits to appear; if the value
16559 being converted can be represented in fewer digits, it shall be expanded with leading
16560 zeros. The default precision shall be 1. The result of converting zero with an explicit
16561 precision of zero shall be no wide characters.

16562 u The **unsigned** argument shall be converted to unsigned decimal format in the style
16563 "dddd". The precision specifies the minimum number of digits to appear; if the value
16564 being converted can be represented in fewer digits, it shall be expanded with leading
16565 zeros. The default precision shall be 1. The result of converting zero with an explicit
16566 precision of zero shall be no wide characters.

16567 x The **unsigned** argument shall be converted to unsigned hexadecimal format in the style
16568 "dddd"; the letters "abcdef" are used. The precision specifies the minimum number
16569 of digits to appear; if the value being converted can be represented in fewer digits, it
16570 shall be expanded with leading zeros. The default precision shall be 1. The result of
16571 converting zero with an explicit precision of zero shall be no wide characters.

16572 X Equivalent to the x conversion specifier, except that letters "ABCDEF" are used instead
16573 of "abcdef".

16574 f, F The **double** argument shall be converted to decimal notation in the style
16575 "[–]*ddd.ddd*", where the number of digits after the radix character shall be equal to
16576 the precision specification. If the precision is missing, it shall be taken as 6; if the
16577 precision is explicitly zero and no '#' flag is present, no radix character shall appear. If
16578 a radix character appears, at least one digit shall appear before it. The value shall be
16579 rounded in an implementation-defined manner to the appropriate number of digits.

16580 A **double** argument representing an infinity shall be converted in one of the styles
16581 "[–]inf" or "[–]infinity"; which style is implementation-defined. A **double**
16582 argument representing a NaN shall be converted in one of the styles "[–]nan" or
16583 "[–]nan(*n-char-sequence*)"; which style, and the meaning of any *n-char-sequence*,
16584 is implementation-defined. The F conversion specifier produces "INF", "INFINITY",
16585 or "NAN" instead of "inf", "infinity", or "nan", respectively.

16586 e, E The **double** argument shall be converted in the style "[–]*d.ddd*e±dd", where there
16587 shall be one digit before the radix character (which is non-zero if the argument is non-
16588 zero) and the number of digits after it shall be equal to the precision; if the precision is
16589 missing, it shall be taken as 6; if the precision is zero and no '#' flag is present, no
16590 radix character shall appear. The value shall be rounded in an implementation-defined
16591 manner to the appropriate number of digits. The E conversion wide character shall
16592 produce a number with 'E' instead of 'e' introducing the exponent. The exponent
16593 shall always contain at least two digits. If the value is zero, the exponent shall be zero.

16594 A **double** argument representing an infinity or NaN shall be converted in the style of
16595 an f or F conversion specifier.

16596 g, G The **double** argument shall be converted in the style f or e (or in the style F or E in the
16597 case of a G conversion specifier), with the precision specifying the number of significant
16598 digits. If an explicit precision is zero, it shall be taken as 1. The style used depends on
16599 the value converted; style e (or E) shall be used only if the exponent resulting from
16600 such a conversion is less than –4 or greater than or equal to the precision. Trailing zeros
16601 shall be removed from the fractional portion of the result; a radix character shall appear
16602 only if it is followed by a digit.

16603 A **double** argument representing an infinity or NaN shall be converted in the style of
16604 an f or F conversion specifier.

a, A A **double** argument representing a floating-point number shall be converted in the style `"[-]0xh.hhhhp±d"`, where there shall be one hexadecimal digit (which is non-zero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point wide character and the number of hexadecimal digits after it shall be equal to the precision; if the precision is missing and FLT_RADIX is a power of 2, then the precision shall be sufficient for an exact representation of the value; if the precision is missing and FLT_RADIX is not a power of 2, then the precision shall be sufficient to distinguish values of type **double**, except that trailing zeros may be omitted; if the precision is zero and the `'#'` flag is not specified, no decimal-point wide character shall appear. The letters `"abcdef"` are used for `a` conversion and the letters `"ABCDEF"` for `A`

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 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.

CX 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*( ).

RETURN VALUE

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 CX 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 CX value, and set *errno* to indicate the error.

ERRORS

For the conditions under which *fwprintf*( ) and *wprintf*( ) fail and may fail, refer to *fputwc*( ).

In addition, all forms of *fwprintf*( ) may fail if:

CX [EILSEQ] A wide-character code that does not correspond to a valid character has been detected.

CX [EINVAL] There are insufficient arguments.

In addition, *fwprintf*( ) and *wprintf*( ) may fail if:

CX [ENOMEM] Insufficient storage space is available.

The *swprintf*( ) shall fail if:

CX [EOVERFLOW] The value of *n* is greater than {INT_MAX} or the number of bytes needed to hold the output excluding the terminating null is greater than {INT_MAX}.

EXAMPLES

To print the language-independent date and time format, the following statement could be used:

```
wprintf(format, weekday, month, day, hour, min);
```

For American usage, *format* could be a pointer to the wide-character string:

```
L"%s, %s %d, %d:%.2d\n"
```

producing the message:

```
Sunday, July 3, 10:02
```

whereas for German usage, *format* could be a pointer to the wide-character string:

```
L"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"
```


16697 producing the message:

16698 ```
Sonntag, 3. Juli, 10:02
```

16699 **APPLICATION USAGE**

16700 None.

16701 **RATIONALE**

16702 None.

16703 **FUTURE DIRECTIONS**

16704 None.

16705 **SEE ALSO**

16706 *btowc( )*, *fputwc( )*, *fwscanf( )*, *mbrtowc( )*, *setlocale( )*, the Base Definitions volume of
16707 IEEE Std 1003.1-200x, Chapter 7, Locale, **<stdio.h>**, **<wchar.h>**

16708 **CHANGE HISTORY**

16709 First released in Issue 5. Included for alignment with ISO/IEC 9899: 1990/Amendment 1: 1995
16710 (E).

16711 **Issue 6**

16712 The Open Group Corrigendum U040/1 is applied to the RETURN VALUE section, describing
16713 the case if *n* or more wide characters are requested to be written using *swprintf( )*.

16714 The normative text is updated to avoid use of the term "must" for application requirements.

16715 The following changes are made for alignment with the ISO/IEC 9899: 1999 standard:

16716 - The prototypes for *fwprintf( )*, *swprintf( )*, and *wprintf( )* are updated.
16717 - The DESCRIPTION is updated.
16718 - The `hh`, `ll`, `j`, `t`, and `z` length modifiers are added.
16719 - The `a`, `A`, and `F` conversion characters are added.
16720 - XSI shading is removed from the description of character string representations of infinity
16721 and NaN floating-point values.

16722 The DESCRIPTION is updated to use the terms "conversion specifier" and "conversion
16723 specification" consistently.

16724 ISO/IEC 9899: 1999 standard, Technical Corrigendum 1 is incorporated.

16725 **Issue 7**

16726 Functionality relating to the `"%n$"` form of conversion specification and the `'''` (apostrophe)
16727 flag is moved from the XSI option to the Base.

16728 The [EOVERFLOW] error is added for *swprintf( )*.
```

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

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 1003.1-200x defers to the ISO C standard.

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.

CX 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, CX and *errno* shall be set to indicate the error.

ERRORS

Refer to *fputc*( ).

EXAMPLES

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 1003.1-200x, **<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*( ).

16775 **NAME**

16776 fwscanf, swscanf, wscanf — convert formatted wide-character input

16777 **SYNOPSIS**

16778 ```
#include <stdio.h>
```
16779 ```
#include <wchar.h>
```

16780 ```
int fwscanf(FILE *restrict stream, const wchar_t *restrict format, ...);
```
16781 ```
int swscanf(const wchar_t *restrict ws,
```
16782 ```
    const wchar_t *restrict format, ...);
```
16783 ```
int wscanf(const wchar_t *restrict format, ... );
```

16784 **DESCRIPTION**

16785 CX The functionality described on this reference page is aligned with the ISO C standard. Any
16786 conflict between the requirements described here and the ISO C standard is unintentional. This
16787 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

16788 The *fwscanf*( ) function shall read from the named input *stream*. The *wscanf*( ) function shall read
16789 from the standard input stream *stdin*. The *swscanf*( ) function shall read from the wide-character
16790 string *ws*. Each function reads wide characters, interprets them according to a format, and stores
16791 the results in its arguments. Each expects, as arguments, a control wide-character string *format*
16792 described below, and a set of *pointer* arguments indicating where the converted input should be
16793 stored. The result is undefined if there are insufficient arguments for the format. If the *format* is
16794 exhausted while arguments remain, the excess arguments are evaluated but are otherwise
16795 ignored.

16796 CX Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than
16797 to the next unused argument. In this case, the conversion specifier wide character % (see below)
16798 is replaced by the sequence "%n$", where *n* is a decimal integer in the range
16799 [1,{NL_ARGMAX}]. This feature provides for the definition of *format* wide-character strings that
16800 select arguments in an order appropriate to specific languages. In *format* wide-character strings
16801 containing the "%*n*$" form of conversion specifications, it is unspecified whether numbered
16802 arguments in the argument list can be referenced from the *format* wide-character string more
16803 than once.

16804 The *format* can contain either form of a conversion specification—that is, % or "%*n*$"— but the
16805 two forms cannot normally be mixed within a single *format* wide-character string. The only
16806 exception to this is that %% or %* can be mixed with the "%*n*$" form. When numbered argument
16807 specifications are used, specifying the *N*th argument requires that all the leading arguments,
16808 from the first to the (*N*–1)th, are pointers.

16809 The *fwscanf*( ) function in all its forms allows for detection of a language-dependent radix
16810 character in the input string, encoded as a wide-character value. The radix character is defined
16811 in the locale of the process (category *LC_NUMERIC*). In the POSIX locale, or in a locale where
16812 the radix character is not defined, the radix character shall default to a period ('.').

16813 The *format* is a wide-character string composed of zero or more directives. Each directive is
16814 composed of one of the following: one or more white-space wide characters (<space>s, <tab>s,
16815 <newline>s, <vertical-tab>s, or <form-feed>s); an ordinary wide character (neither '%' nor a
16816 white-space character); or a conversion specification.

16817 CX Each conversion specification is introduced by the '%' or by the character sequence "%*n*$",
16818 after which the following appear in sequence:

16819 • 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.
- 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
CX conversion result if the conversion specification is introduced by `%`, or in the *n*th argument if introduced by the wide-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**.

16867 `ll` (ell-ell)
16868 Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an
16869 argument with type pointer to **long long** or **unsigned long long**.

16870 `j` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an
16871 argument with type pointer to **intmax_t** or **uintmax_t**.

16872 `z` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an
16873 argument with type pointer to **size_t** or the corresponding signed integer type.

16874 `t` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an
16875 argument with type pointer to **ptrdiff_t** or the corresponding **unsigned** type.

16876 `L` Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to an
16877 argument with type pointer to **long double**.

16878 If a length modifier appears with any conversion specifier other than as specified above, the
16879 behavior is undefined.

16880 The following conversion specifier wide characters are valid:

16881 `d` Matches an optionally signed decimal integer, whose format is the same as expected for
16882 the subject sequence of *wcstol*( ) with the value 10 for the *base* argument. In the absence
16883 of a size modifier, the application shall ensure that the corresponding argument is a
16884 pointer to **int**.

16885 `i` Matches an optionally signed integer, whose format is the same as expected for the
16886 subject sequence of *wcstol*( ) with 0 for the *base* argument. In the absence of a size
16887 modifier, the application shall ensure that the corresponding argument is a pointer to
16888 **int**.

16889 `o` Matches an optionally signed octal integer, whose format is the same as expected for
16890 the subject sequence of *wcstoul*( ) with the value 8 for the *base* argument. In the absence
16891 of a size modifier, the application shall ensure that the corresponding argument is a
16892 pointer to **unsigned**.

16893 `u` Matches an optionally signed decimal integer, whose format is the same as expected for
16894 the subject sequence of *wcstoul*( ) with the value 10 for the *base* argument. In the absence
16895 of a size modifier, the application shall ensure that the corresponding argument is a
16896 pointer to **unsigned**.

16897 `x` Matches an optionally signed hexadecimal integer, whose format is the same as
16898 expected for the subject sequence of *wcstoul*( ) with the value 16 for the *base* argument.
16899 In the absence of a size modifier, the application shall ensure that the corresponding
16900 argument is a pointer to **unsigned**.

16901 `a`, `e`, `f`, `g`
16902 Matches an optionally signed floating-point number, infinity, or NaN whose format is
16903 the same as expected for the subject sequence of *wcstod*( ). In the absence of a size
16904 modifier, the application shall ensure that the corresponding argument is a pointer to
16905 **float**.

16906 If the *fwprintf*( ) family of functions generates character string representations for
16907 infinity and NaN (a symbolic entity encoded in floating-point format) to support
16908 IEEE Std 754-1985, the *fwscanf*( ) family of functions shall recognize them as input.

16909 `s` Matches a sequence of non white-space wide characters. If no `l` (ell) qualifier is present,
16910 characters from the input field shall be converted as if by repeated calls to the
16911 *wcrtomb*( ) function, with the conversion state described by an **mbstate_t** object
16912 initialized to zero before the first wide character is converted. The application shall
16913 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 `l` (ell) qualifier is present, wide 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.

If an `l` (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 `l` (ell) length modifier 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 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 `l` (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 *fwprintf*( ) 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.

16961 n No input is consumed. The application shall ensure that the corresponding argument is
16962 a pointer to the integer into which is to be written the number of wide characters read
16963 from the input so far by this call to the *fwscanf*( ) functions. Execution of a %n
16964 conversion specification shall not increment the assignment count returned at the
16965 completion of execution of the function. No argument shall be converted, but one shall
16966 be consumed. If the conversion specification includes an assignment-suppressing wide
16967 character or a field width, the behavior is undefined.

16968 XSI C Equivalent to lc.

16969 XSI S Equivalent to ls.

16970 % Matches a single '%' wide character; no conversion or assignment shall occur. The
16971 complete conversion specification shall be %%.

16972 If a conversion specification is invalid, the behavior is undefined.

16973 The conversion specifiers A, E, F, G, and X are also valid and shall be equivalent to, respectively,
16974 a, e, f, g, and x.

16975 If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before
16976 any wide characters matching the current conversion specification (except for %n) have been
16977 read (other than leading white-space, where permitted), execution of the current conversion
16978 specification shall terminate with an input failure. Otherwise, unless execution of the current
16979 conversion specification is terminated with a matching failure, execution of the following
16980 conversion specification (if any) shall be terminated with an input failure.

16981 Reaching the end of the string in *swscanf*( ) shall be equivalent to encountering end-of-file for
16982 *fwscanf*( ).

16983 If conversion terminates on a conflicting input, the offending input shall be left unread in the
16984 input. Any trailing white space (including <newline>) shall be left unread unless matched by a
16985 conversion specification. The success of literal matches and suppressed assignments is only
16986 directly determinable via the %n conversion specification.

16987 CX The *fwscanf*( ) and *wscanf*( ) functions may mark the *st_atime* field of the file associated with
16988 *stream* for update. The *st_atime* field shall be marked for update by the first successful execution
16989 of *fgetc*( ), *fgetwc*( ), *fgets*( ), *fgetws*( ), *fread*( ), *getc*( ), *getwc*( ), *getchar*( ), *getwchar*( ), *gets*( ), *fscanf*( ),
16990 or *fwscanf*( ) using *stream* that returns data not supplied by a prior call to *ungetc*( ).

16991 **RETURN VALUE**

16992 Upon successful completion, these functions shall return the number of successfully matched
16993 and assigned input items; this number can be zero in the event of an early matching failure. If
16994 the input ends before the first matching failure or conversion, EOF shall be returned. If a read
16995 CX error occurs, the error indicator for the stream is set, EOF shall be returned, and *errno* shall be
16996 set to indicate the error.

16997 **ERRORS**

16998 For the conditions under which the *fwscanf*( ) functions shall fail and may fail, refer to *fgetwc*( ).

16999 In addition, *fwscanf*( ) may fail if:

17000 CX [EILSEQ] Input byte sequence does not form a valid character.

17001 CX [EINVAL] There are insufficient arguments.

EXAMPLES

The call:

```
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:

```
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 1003.1-200x, Chapter 7, Locale, **<langinfo.h>**, **<stdio.h>**, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

The normative text 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.

Issue 7

Functionality relating to the `"%n$"` form of conversion specification is moved from the XSI option to the Base.

17042 **NAME**

17043 gai_strerror — address and name information error description

17044 **SYNOPSIS**

17045 `#include <netdb.h>`

17046 `const char *gai_strerror(int ecode);`

17047 **DESCRIPTION**

17048 The *gai_strerror*( ) function shall return a text string describing an error value for the *getaddrinfo*( )
17049 and *getnameinfo*( ) functions listed in the **<netdb.h>** header.

17050 When the *ecode* argument is one of the following values listed in the **<netdb.h>** header:

17051 [EAI_AGAIN] [EAI_NONAME]
17052 [EAI_BADFLAGS] [EAI_OVERFLOW]
17053 [EAI_FAIL] [EAI_SERVICE]
17054 [EAI_FAMILY] [EAI_SOCKTYPE]
17055 [EAI_MEMORY] [EAI_SYSTEM]

17056 the function return value shall point to a string describing the error. If the argument is not one
17057 of those values, the function shall return a pointer to a string whose contents indicate an
17058 unknown error.

17059 **RETURN VALUE**

17060 Upon successful completion, *gai_strerror*( ) shall return a pointer to an implementation-defined
17061 string.

17062 **ERRORS**

17063 No errors are defined.

17064 **EXAMPLES**

17065 None.

17066 **APPLICATION USAGE**

17067 None.

17068 **RATIONALE**

17069 None.

17070 **FUTURE DIRECTIONS**

17071 None.

17072 **SEE ALSO**

17073 *getaddrinfo*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<netdb.h>**

17074 **CHANGE HISTORY**

17075 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

17076 The Open Group Base Resolution bwg2001-009 is applied, which changes the return type from
17077 **char** * to **const char** *. This is for coordination with the IPnG Working Group.

17078 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/22 is applied, adding the
17079 [EAI_OVERFLOW] error code.

17080 **NAME**

17081 getaddrinfo — get address information

17082 **SYNOPSIS**

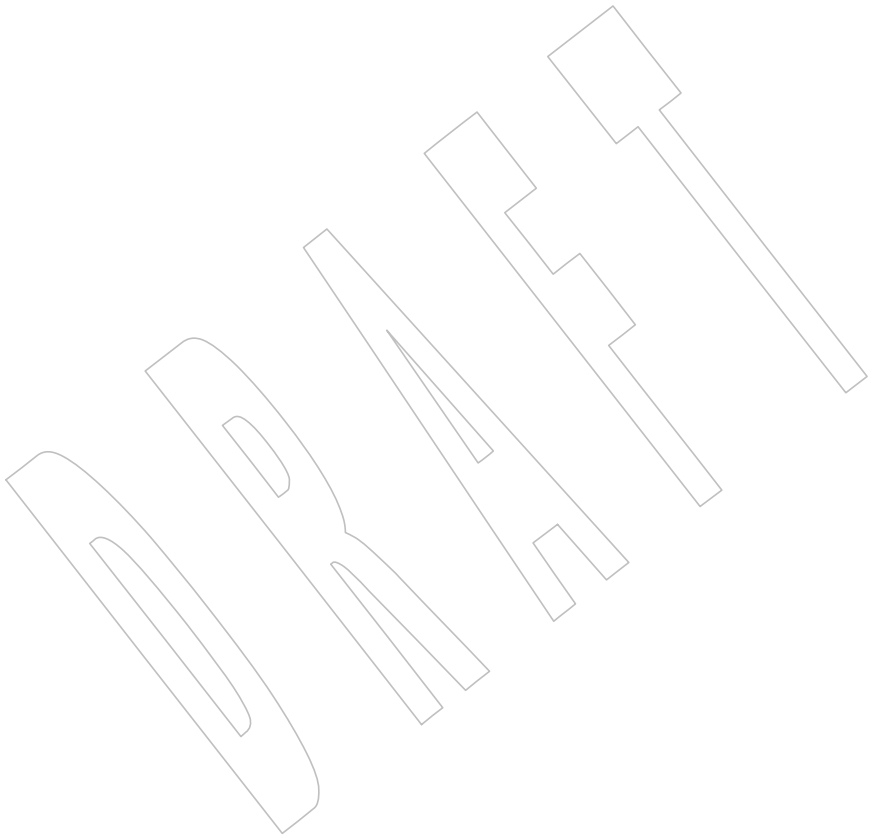
17083 ```
#include <sys/socket.h>
```

17084 ```
#include <netdb.h>
```

17085 ```
int getaddrinfo(const char *restrict nodename,
```

17086 ```
    const char *restrict servname,
```

17087 ```
    const struct addrinfo *restrict hints,
```

17088 ```
    struct addrinfo **restrict res);
```

17089 **DESCRIPTION**

17090 Refer to *freeaddrinfo( )*.



17091 **NAME**

17092 getc — get a byte from a stream

17093 **SYNOPSIS**

17094
```
#include <stdio.h>
```

17095
```
int getc(FILE *stream);
```

17096 **DESCRIPTION**

17097 CX The functionality described on this reference page is aligned with the ISO C standard. Any
17098 conflict between the requirements described here and the ISO C standard is unintentional. This
17099 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

17100 The *getc*( ) function shall be equivalent to *fgetc*( ), except that if it is implemented as a macro it
17101 may evaluate *stream* more than once, so the argument should never be an expression with side
17102 effects.

17103 **RETURN VALUE**

17104 Refer to *fgetc*( ).

17105 **ERRORS**

17106 Refer to *fgetc*( ).

17107 **EXAMPLES**

17108 None.

17109 **APPLICATION USAGE**

17110 If the integer value returned by *getc*( ) is stored into a variable of type **char** and then compared
17111 against the integer constant EOF, the comparison may never succeed, because sign-extension of
17112 a variable of type **char** on widening to integer is implementation-defined.

17113 Since it may be implemented as a macro, *getc*( ) may treat incorrectly a *stream* argument with
17114 side effects. In particular, *getc*(*f*++) does not necessarily work as expected. Therefore, use of this
17115 function should be preceded by `"#undef getc"` in such situations; *fgetc*( ) could also be used.

17116 **RATIONALE**

17117 None.

17118 **FUTURE DIRECTIONS**

17119 None.

17120 **SEE ALSO**

17121 *fgetc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

17122 **CHANGE HISTORY**

17123 First released in Issue 1. Derived from Issue 1 of the SVID.

17124 **NAME**

17125 getc_unlocked, getchar_unlocked, putc_unlocked, putchar_unlocked — stdio with explicit client
17126 locking

17127 **SYNOPSIS**

17128 CX
```
#include <stdio.h>

int getc_unlocked(FILE *stream);
int getchar_unlocked(void);
int putc_unlocked(int c, FILE *stream);
int putchar_unlocked(int c);
```

17133 **DESCRIPTION**

17134 Versions of the functions *getc*( ), *getchar*( ), *putc*( ), and *putchar*( ) respectively named
17135 *getc_unlocked*( ), *getchar_unlocked*( ), *putc_unlocked*( ), and *putchar_unlocked*( ) shall be provided
17136 which are functionally equivalent to the original versions, with the exception that they are not
17137 required to be implemented in a thread-safe manner. They may only safely be used within a
17138 scope protected by *flockfile*( ) (or *ftrylockfile*( )) and *funlockfile*( ). These functions may safely be
17139 used in a multi-threaded program if and only if they are called while the invoking thread owns
17140 the (**FILE** *) object, as is the case after a successful call to the *flockfile*( ) or *ftrylockfile*( ) functions.

17141 **RETURN VALUE**

17142 See *getc*( ), *getchar*( ), *putc*( ), and *putchar*( ).

17143 **ERRORS**

17144 See *getc*( ), *getchar*( ), *putc*( ), and *putchar*( ).

17145 **EXAMPLES**

17146 None.

17147 **APPLICATION USAGE**

17148 Since they may be implemented as macros, *getc_unlocked*( ) and *putc_unlocked*( ) may treat
17149 incorrectly a *stream* argument with side effects. In particular, *getc_unlocked*(*f++) and
17150 *putc_unlocked*(*f++) do not necessarily work as expected. Therefore, use of these functions in
17151 such situations should be preceded by the following statement as appropriate:

```
17152 #undef getc_unlocked
17153 #undef putc_unlocked
```

17154 **RATIONALE**

17155 Some I/O functions are typically implemented as macros for performance reasons (for example,
17156 *putc*( ) and *getc*( )). For safety, they need to be synchronized, but it is often too expensive to
17157 synchronize on every character. Nevertheless, it was felt that the safety concerns were more
17158 important; consequently, the *getc*( ), *getchar*( ), *putc*( ), and *putchar*( ) functions are required to be
17159 thread-safe. However, unlocked versions are also provided with names that clearly indicate the
17160 unsafe nature of their operation but can be used to exploit their higher performance. These
17161 unlocked versions can be safely used only within explicitly locked program regions, using
17162 exported locking primitives. In particular, a sequence such as:

```
17163 flockfile(fileptr);
17164 putc_unlocked('1', fileptr);
17165 putc_unlocked('\n', fileptr);
17166 fprintf(fileptr, "Line 2\n");
17167 funlockfile(fileptr);
```

17168 is permissible, and results in the text sequence:

```
1
Line 2
```

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 1003.1-200x, **<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.

Issue 7

The *getc_unlocked*( ), *getchar_unlocked*( ), *putc_unlocked*( ), and *putchar_unlocked*( ) functions are moved from the Thread-Safe Functions option to the Base.

17206 **NAME**

17207 getchar — get a byte from a stdin stream

17208 **SYNOPSIS**

17209
```
#include <stdio.h>
```

17210
```
int getchar(void);
```

17211 **DESCRIPTION**

17212 CX The functionality described on this reference page is aligned with the ISO C standard. Any
17213 conflict between the requirements described here and the ISO C standard is unintentional. This
17214 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

17215 The *getchar*( ) function shall be equivalent to *getc*(*stdin*).

17216 **RETURN VALUE**

17217 Refer to *fgetc*( ).

17218 **ERRORS**

17219 Refer to *fgetc*( ).

17220 **EXAMPLES**

17221 None.

17222 **APPLICATION USAGE**

17223 If the integer value returned by *getchar*( ) is stored into a variable of type **char** and then
17224 compared against the integer constant EOF, the comparison may never succeed, because sign-
17225 extension of a variable of type **char** on widening to integer is implementation-defined.

17226 **RATIONALE**

17227 None.

17228 **FUTURE DIRECTIONS**

17229 None.

17230 **SEE ALSO**

17231 *getc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

17232 **CHANGE HISTORY**

17233 First released in Issue 1. Derived from Issue 1 of the SVID.

17234 **NAME**

17235 getchar_unlocked — stdio with explicit client locking

17236 **SYNOPSIS**

17237 CX
```
#include <stdio.h>

int getchar_unlocked(void);
```

17239 **DESCRIPTION**

17240 Refer to *getc_unlocked( )*.

17241 **NAME**

17242 getcwd — get the pathname of the current working directory

17243 **SYNOPSIS**

17244
```
#include <unistd.h>
```

17245
```
char *getcwd(char *buf, size_t size);
```

17246 **DESCRIPTION**

17247 The *getcwd*( ) function shall place an absolute pathname of the current working directory in the
17248 array pointed to by *buf*, and return *buf*. The pathname copied to the array shall contain no
17249 components that are symbolic links. The *size* argument is the size in bytes of the character array
17250 pointed to by the *buf* argument. If *buf* is a null pointer, the behavior of *getcwd*( ) is unspecified.

17251 **RETURN VALUE**

17252 Upon successful completion, *getcwd*( ) shall return the *buf* argument. Otherwise, *getcwd*( ) shall
17253 return a null pointer and set *errno* to indicate the error. The contents of the array pointed to by
17254 *buf* are then undefined.

17255 **ERRORS**

17256 The *getcwd*( ) function shall fail if:

17257 [EINVAL] The *size* argument is 0.

17258 [ERANGE] The *size* argument is greater than 0, but is smaller than the length of the
17259 pathname +1.

17260 The *getcwd*( ) function may fail if:

17261 [EACCES] Read or search permission was denied for a component of the pathname.

17262 [ENOMEM] Insufficient storage space is available.

17263 **EXAMPLES**

17264 **Determining the Absolute Pathname of the Current Working Directory**

17265 The following example returns a pointer to an array that holds the absolute pathname of the
17266 current working directory. The pointer is returned in the *ptr* variable, which points to the *buf*
17267 array where the pathname is stored.

17268
```
#include <stdlib.h>
```
17269
```
#include <unistd.h>
```
17270
```
...
```
17271
```
long size;
```
17272
```
char *buf;
```
17273
```
char *ptr;
```

17274
```
size = pathconf(".", _PC_PATH_MAX);
```

17275
```
if ((buf = (char *)malloc((size_t)size)) != NULL)
```
17276
```
    ptr = getcwd(buf, (size_t)size);
```
17277
```
...
```

17278 **APPLICATION USAGE**

17279 None.

17280 **RATIONALE**

17281 Since the maximum pathname length is arbitrary unless {PATH_MAX} is defined, an application
17282 generally cannot supply a *buf* with *size* {{PATH_MAX}+1}.

17283 Having *getcwd*( ) take no arguments and instead use the *malloc*( ) function to produce space for
17284 the returned argument was considered. The advantage is that *getcwd*( ) knows how big the
17285 working directory pathname is and can allocate an appropriate amount of space. But the
17286 programmer would have to use the *free*( ) function to free the resulting object, or each use of
17287 *getcwd*( ) would further reduce the available memory. Also, *malloc*( ) and *free*( ) are used nowhere
17288 else in this volume of IEEE Std 1003.1-200x. Finally, *getcwd*( ) is taken from the SVID where it has
17289 the two arguments used in this volume of IEEE Std 1003.1-200x.

17290 The older function *getwd*( ) was rejected for use in this context because it had only a buffer
17291 argument and no *size* argument, and thus had no way to prevent overwriting the buffer, except
17292 to depend on the programmer to provide a large enough buffer.

17293 On some implementations, if *buf* is a null pointer, *getcwd*( ) may obtain *size* bytes of memory
17294 using *malloc*( ). In this case, the pointer returned by *getcwd*( ) may be used as the argument in a
17295 subsequent call to *free*( ). Invoking *getcwd*( ) with *buf* as a null pointer is not recommended in
17296 conforming applications.

17297 If a program is operating in a directory where some (grand)parent directory does not permit
17298 reading, *getcwd*( ) may fail, as in most implementations it must read the directory to determine
17299 the name of the file. This can occur if search, but not read, permission is granted in an
17300 intermediate directory, or if the program is placed in that directory by some more privileged
17301 process (for example, login). Including the [EACCES] error condition makes the reporting of the
17302 error consistent and warns the application writer that *getcwd*( ) can fail for reasons beyond the
17303 control of the application writer or user. Some implementations can avoid this occurrence (for
17304 example, by implementing *getcwd*( ) using *pwd*, where *pwd* is a set-user-root process), thus the
17305 error was made optional. Since this volume of IEEE Std 1003.1-200x permits the addition of
17306 other errors, this would be a common addition and yet one that applications could not be
17307 expected to deal with without this addition.

17308 **FUTURE DIRECTIONS**

17309 None.

17310 **SEE ALSO**

17311 *malloc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

17312 **CHANGE HISTORY**

17313 First released in Issue 1. Derived from Issue 1 of the SVID.

17314 **Issue 6**

17315 The following new requirements on POSIX implementations derive from alignment with the
17316 Single UNIX Specification:

17317 • The [ENOMEM] optional error condition is added.

17318 **NAME**

17319 getdate — convert user format date and time

17320 **SYNOPSIS**

17321 XSI
```
#include <time.h>

struct tm *getdate(const char *string);
```

17323 **DESCRIPTION**

17324 The *getdate*( ) function shall convert a string representation of a date or time into a broken-down
17325 time.

17326 The external variable or macro *getdate_err* is used by *getdate*( ) to return error values.

17327 Templates are used to parse and interpret the input string. The templates are contained in a text
17328 file identified by the environment variable *DATEMSK*. The *DATEMSK* variable should be set to
17329 indicate the full pathname of the file that contains the templates. The first line in the template
17330 that matches the input specification is used for interpretation and conversion into the internal
17331 time format.

17332 The following conversion specifications shall be supported:

17333 `%%` Equivalent to %.

17334 `%a` Abbreviated weekday name.

17335 `%A` Full weekday name.

17336 `%b` Abbreviated month name.

17337 `%B` Full month name.

17338 `%c` Locale's appropriate date and time representation.

17339 `%C` Century number [00,99]; leading zeros are permitted but not required.

17340 `%d` Day of month [01,31]; the leading 0 is optional.

17341 `%D` Date as `%m/%d/%y`.

17342 `%e` Equivalent to `%d`.

17343 `%h` Abbreviated month name.

17344 `%H` Hour [00,23].

17345 `%I` Hour [01,12].

17346 `%m` Month number [01,12].

17347 `%M` Minute [00,59].

17348 `%n` Equivalent to <newline>.

17349 `%p` Locale's equivalent of either AM or PM.

17350 `%r` The locale's appropriate representation of time in AM and PM notation. In the POSIX
17351 locale, this shall be equivalent to `%I:%M:%S %p`.

17352 `%R` Time as `%H:%M`.

17353 `%S` Seconds [00,60]. The range goes to 60 (rather than stopping at 59) to allow positive leap
17354 seconds to be expressed. Since leap seconds cannot be predicted by any algorithm, leap
17355 second data must come from some external source.

17356 %t Equivalent to <tab>.

17357 %T Time as %H:%M:%S.

17358 %w Weekday number (Sunday = [0,6]).

17359 %x Locale's appropriate date representation.

17360 %X Locale's appropriate time representation.

17361 %y Year within century. When a century is not otherwise specified, values in the range
17362 [69,99] shall refer to years 1969 to 1999 inclusive, and values in the range [00,68] shall
17363 refer to years 2000 to 2068 inclusive.

17364 **Note:** It is expected that in a future version of IEEE Std 1003.1-200x the default century
17365 inferred from a 2-digit year will change. (This would apply to all commands
17366 accepting a 2-digit year as input.)

17367 %Y Year as "ccyy" (for example, 2001).

17368 %Z Timezone name or no characters if no timezone exists. If the timezone supplied by %Z is
17369 not the timezone that *getdate*( ) expects, an invalid input specification error shall result.
17370 The *getdate*( ) function calculates an expected timezone based on information supplied
17371 to the function (such as the hour, day, and month).

17372 The match between the template and input specification performed by *getdate*( ) shall be case-
17373 insensitive.

17374 The month and weekday names can consist of any combination of upper and lowercase letters.
17375 The process can request that the input date or time specification be in a specific language by
17376 setting the *LC_TIME* category (see *setlocale*( )).

17377 Leading zeros are not necessary for the descriptors that allow leading zeros. However, at most
17378 two digits are allowed for those descriptors, including leading zeros. Extra whitespace in either
17379 the template file or in *string* shall be ignored.

17380 The results are undefined if the conversion specifications %c, %x, and %X include unsupported
17381 conversion specifications.

17382 The following rules apply for converting the input specification into the internal format:

17383 • If %Z is being scanned, then *getdate*( ) shall initialize the broken-down time to be the current
17384 time in the scanned timezone. Otherwise, it shall initialize the broken-down time based on
17385 the current local time as if *localtime*( ) had been called.

17386 • If only the weekday is given, the day chosen shall be the day, starting with today and
17387 moving into the future, which first matches the named day.

17388 • If only the month (and no year) is given, the month chosen shall be the month, starting
17389 with the current month and moving into the future, which first matches the named month.
17390 The first day of the month shall be assumed if no day is given.

17391 • If no hour, minute, and second are given, the current hour, minute, and second shall be
17392 assumed.

17393 • If no date is given, the hour chosen shall be the hour, starting with the current hour and
17394 moving into the future, which first matches the named hour.

17395 If a conversion specification in the DATEMSK file does not correspond to one of the conversion
17396 specifications above, the behavior is unspecified.

17397 The *getdate*( ) function need not be thread-safe. A function that is not required to be thread-safe is
17398 not required to be reentrant.

RETURN VALUE

Upon successful completion, *getdate*( ) shall return a pointer to a **struct tm**. Otherwise, it shall return a null pointer and set *getdate_err* to indicate the error.

ERRORS

The *getdate*( ) function shall fail in the following cases, setting *getdate_err* to the value shown in the list below. Any changes to *errno* are unspecified.

1. The *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 **time_t** (representing the time in seconds since the Epoch).

EXAMPLES

1. The following example shows the possible contents of a template:

```
%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
```

2. The following are examples of valid input specifications for the template in Example 1:

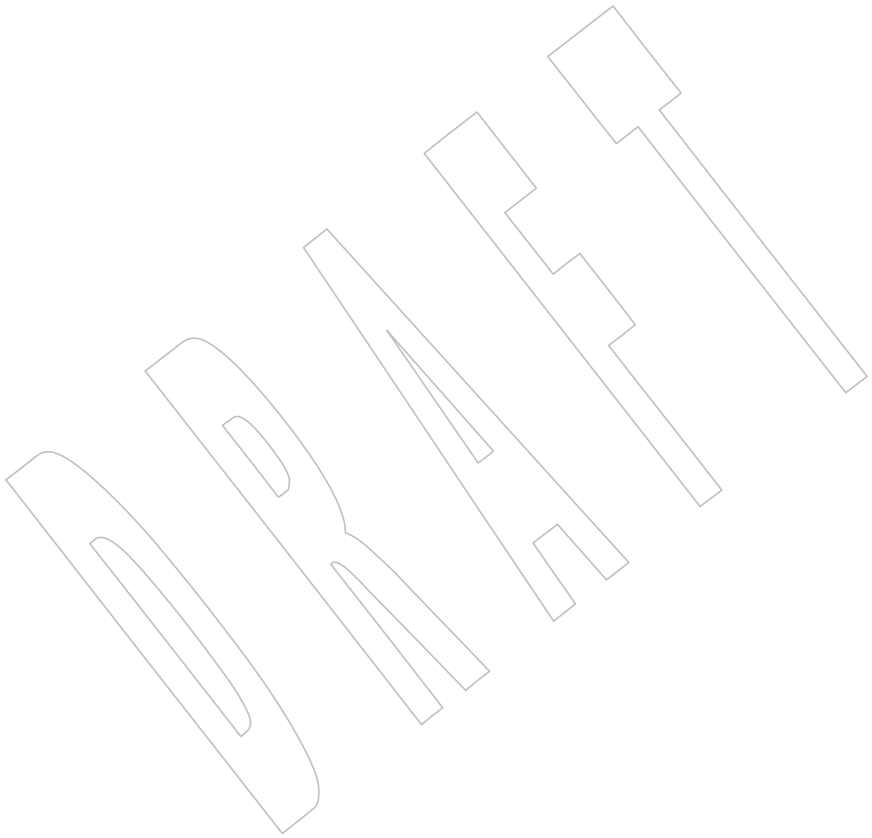
```
getdate("10/1/87 4 PM");
getdate("Friday");
getdate("Friday September 18, 1987, 10:30:30");
getdate("24,9,1986 10:30");
getdate("at monday the 1st of december in 1986");
getdate("run job at 3 PM, december 2nd");
```

   If the *LC_TIME* category is set to a German locale that includes *freitag* as a weekday name and *oktober* as a month name, the following would be valid:

```
getdate("freitag den 10. oktober 1986 10.30 Uhr");
```

3. The following example shows how local date and time specification can be defined in the template:

Invocation	Line in Template
`getdate("11/27/86")`	`%m/%d/%y`
`getdate("27.11.86")`	`%d.%m.%y`
`getdate("86-11-27")`	`%y-%m-%d`
`getdate("Friday 12:00:00")`	`%A %H:%M:%S`



17488 The DESCRIPTION is updated to refer to conversion specifications instead of field descriptors
17489 for consistency with other functions.

NAME

getdelim, getline — read a delimited record from *stream*

SYNOPSIS

CX
```
#include <stdio.h>

ssize_t getdelim(char **lineptr, size_t *n, int delimiter,
    FILE *stream);
ssize_t getline(char **lineptr, size_t *n, FILE *stream);
```

DESCRIPTION

The *getdelim*( ) function shall read from *stream* until it encounters a character matching the *delimiter* character. The *delimiter* argument is an **int**, the value of which the application shall ensure is a character representable as an **unsigned char** of equal value that terminates the read process. If the *delimiter* argument has any other value, the behavior is undefined.

The application shall ensure that **lineptr* is a valid argument that could be passed to the *free*( ) function. If **n* is non-zero, the application shall ensure that **lineptr* points to an object of size at least **n* bytes.

The size of the object pointed to by **lineptr* shall be increased to fit the incoming line, if it isn't already large enough. The characters read shall be stored in the string pointed to by the *lineptr* argument.

The *getline*( ) function shall be equivalent to the *getdelim*( ) function with the *delimiter* character equal to the <newline> character.

RETURN VALUE

Upon successful completion, the *getdelim*( ) function shall return the number of characters written into the buffer, including the delimiter character if one was encountered before EOF. Otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

These functions shall fail if:

[EINVAL] When *lineptr* or *n* are a null pointer.

[ENOMEM] Insufficient memory is available.

These functions may fail if:

[EINVAL] *stream* is not a valid file descriptor.

[EOVERFLOW] More than {SSIZE_MAX} characters were read without encountering the *delimiter* character.

EXAMPLES

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

int main(void)
{
    FILE *fp;
    char *line = NULL;
    size_t len = 0;
    ssize_t read;
    fp = fopen("/etc/motd", "r");
    if (fp == NULL)
```



```
            exit(1);
        while ((read = getline(&line, &len, fp)) != -1) {
            printf("Retrieved line of length %zu :\n", read);
            printf("%s", line);
        }
        if (line)
            free(line);
        fclose(fp);
        return 0;
    }
```

APPLICATION USAGE

Setting **lineptr* to a null pointer and **n* to zero are allowed and a recommended way to start parsing a file.

RATIONALE

These functions are widely used to solve the problem that the *fgets*( ) function has with long lines. The functions automatically enlarge the target buffers if needed. These are especially useful since they reduce code needed for applications.

FUTURE DIRECTIONS

None.

SEE ALSO

fgets( ), *free*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 7.

17556 **NAME**

17557 getegid — get the effective group ID

17558 **SYNOPSIS**

17559
```
#include <unistd.h>
```

17560
```
gid_t getegid(void);
```

17561 **DESCRIPTION**

17562 The *getegid*( ) function shall return the effective group ID of the calling process.

17563 **RETURN VALUE**

17564 The *getegid*( ) function shall always be successful and no return value is reserved to indicate an
17565 error.

17566 **ERRORS**

17567 No errors are defined.

17568 **EXAMPLES**

17569 None.

17570 **APPLICATION USAGE**

17571 None.

17572 **RATIONALE**

17573 None.

17574 **FUTURE DIRECTIONS**

17575 None.

17576 **SEE ALSO**

17577 *geteuid*( ), *getgid*( ), *getuid*( ), *setegid*( ), *seteuid*( ), *setgid*( ), *setregid*( ), *setreuid*( ), *setuid*( ), the Base
17578 Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

17579 **CHANGE HISTORY**

17580 First released in Issue 1. Derived from Issue 1 of the SVID.

17581 **Issue 6**

17582 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

17583 The following new requirements on POSIX implementations derive from alignment with the
17584 Single UNIX Specification:

17585 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
17586 required for conforming implementations of previous POSIX specifications, it was not
17587 required for UNIX applications.

17588 **NAME**

17589 getenv — get value of an environment variable

17590 **SYNOPSIS**

17591
```
#include <stdlib.h>
```

17592
```
char *getenv(const char *name);
```

17593 **DESCRIPTION**

17594 CX The functionality described on this reference page is aligned with the ISO C standard. Any
17595 conflict between the requirements described here and the ISO C standard is unintentional. This
17596 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

17597 The *getenv*( ) function shall search the environment of the calling process (see the Base
17598 Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables) for the
17599 environment variable *name* if it exists and return a pointer to the value of the environment
17600 variable. If the specified environment variable cannot be found, a null pointer shall be returned.
17601 The application shall ensure that it does not modify the string pointed to by the *getenv*( )
17602 function.

17603 CX The string pointed to may be overwritten by a subsequent call to *getenv*( ), *setenv*( ), *unsetenv*( ),
17604 XSI or *putenv*( ) but shall not be overwritten by a call to any other function in this volume of
17605 IEEE Std 1003.1-200x.

17606 CX If the application modifies *environ* or the pointers to which it points, the behavior of *getenv*( ) is
17607 undefined.

17608 The *getenv*( ) function need not be thread-safe. A function that is not required to be thread-safe is
17609 not required to be reentrant.

17610 **RETURN VALUE**

17611 Upon successful completion, *getenv*( ) shall return a pointer to a string containing the *value* for
17612 the specified *name*. If the specified *name* cannot be found in the environment of the calling
17613 process, a null pointer shall be returned.

17614 **ERRORS**

17615 No errors are defined.

17616 **EXAMPLES**

17617 **Getting the Value of an Environment Variable**

17618 The following example gets the value of the *HOME* environment variable.

17619
```
#include <stdlib.h>
```
17620
```
...
```
17621
```
const char *name = "HOME";
```
17622
```
char *value;
```

17623
```
value = getenv(name);
```

17624 **APPLICATION USAGE**

17625 None.

17626 **RATIONALE**

17627 The *clearenv*( ) function was considered but rejected. The *putenv*( ) function has now been
17628 included for alignment with the Single UNIX Specification.

17629 The *getenv*( ) function is inherently not reentrant because it returns a value pointing to static
17630 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 1003.1-200x 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 1003.1-200x, 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 normative text is updated to avoid use of the term "must" for application requirements.

Issue 7

Austin Group Interpretation 1003.1-2001 #062 is applied, clarifying that a call to *putenv*( ) may also cause the string to be overwritten.

17679 **NAME**

17680 geteuid — get the effective user ID

17681 **SYNOPSIS**

17682
```
#include <unistd.h>
```

17683
```
uid_t geteuid(void);
```

17684 **DESCRIPTION**

17685 The *geteuid*( ) function shall return the effective user ID of the calling process.

17686 **RETURN VALUE**

17687 The *geteuid*( ) function shall always be successful and no return value is reserved to indicate an
17688 error.

17689 **ERRORS**

17690 No errors are defined.

17691 **EXAMPLES**

17692 None.

17693 **APPLICATION USAGE**

17694 None.

17695 **RATIONALE**

17696 None.

17697 **FUTURE DIRECTIONS**

17698 None.

17699 **SEE ALSO**

17700 *getegid*( ), *getgid*( ), *getuid*( ), *setegid*( ), *seteuid*( ), *setgid*( ), *setregid*( ), *setreuid*( ), *setuid*( ), the Base
17701 Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

17702 **CHANGE HISTORY**

17703 First released in Issue 1. Derived from Issue 1 of the SVID.

17704 **Issue 6**

17705 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

17706 The following new requirements on POSIX implementations derive from alignment with the
17707 Single UNIX Specification:

17708 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
17709 required for conforming implementations of previous POSIX specifications, it was not
17710 required for UNIX applications.

17711 **NAME**

17712 getgid — get the real group ID

17713 **SYNOPSIS**

17714
```
#include <unistd.h>
```

17715
```
gid_t getgid(void);
```

17716 **DESCRIPTION**

17717 The *getgid*( ) function shall return the real group ID of the calling process.

17718 **RETURN VALUE**

17719 The *getgid*( ) function shall always be successful and no return value is reserved to indicate an
17720 error.

17721 **ERRORS**

17722 No errors are defined.

17723 **EXAMPLES**

17724 None.

17725 **APPLICATION USAGE**

17726 None.

17727 **RATIONALE**

17728 None.

17729 **FUTURE DIRECTIONS**

17730 None.

17731 **SEE ALSO**

17732 *getegid*( ), *geteuid*( ), *getuid*( ), *setegid*( ), *seteuid*( ), *setgid*( ), *setregid*( ), *setreuid*( ), *setuid*( ), the Base
17733 Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

17734 **CHANGE HISTORY**

17735 First released in Issue 1. Derived from Issue 1 of the SVID.

17736 **Issue 6**

17737 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

17738 The following new requirements on POSIX implementations derive from alignment with the
17739 Single UNIX Specification:

17740 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
17741 required for conforming implementations of previous POSIX specifications, it was not
17742 required for UNIX applications.

17743 **NAME**

17744 getgrent — get the group database entry

17745 **SYNOPSIS**

17746 XSI
```
#include <grp.h>

struct group *getgrent(void);
```
17747

17748 **DESCRIPTION**

17749 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 thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

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] All file descriptors available to the process are currently open.

[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.

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.

```
#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 1003.1-200x, **<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.

Issue 7

SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

The *getgrgid_r*( ) function is moved from the Thread-Safe Functions option to the Base.

17843 **NAME**

17844 getgrnam, getgrnam_r — search group database for a name

17845 **SYNOPSIS**

17846
```
#include <grp.h>
```

17847
```
struct group *getgrnam(const char *name);
```
17848
```
int getgrnam_r(const char *name, struct group *grp, char *buffer,
```
17849
```
    size_t bufsize, struct group **result);
```

17850 **DESCRIPTION**

17851 The *getgrnam*( ) function shall search the group database for an entry with a matching *name*.

17852 The *getgrnam*( ) function need not be thread-safe. A function that is not required to be thread-
17853 safe is not required to be reentrant.

17854 The *getgrnam_r*( ) function shall update the **group** structure pointed to by *grp* and store a pointer
17855 to that structure at the location pointed to by *result*. The structure shall contain an entry from
17856 the group database with a matching *gid* or *name*. Storage referenced by the **group** structure is
17857 allocated from the memory provided with the *buffer* parameter, which is *bufsize* bytes in size. The
17858 maximum size needed for this buffer can be determined with the {_SC_GETGR_R_SIZE_MAX}
17859 *sysconf*( ) parameter. A NULL pointer is returned at the location pointed to by *result* on error or if
17860 the requested entry is not found.

17861 **RETURN VALUE**

17862 The *getgrnam*( ) function shall return a pointer to a **struct group** with the structure defined in
17863 **<grp.h>** with a matching entry if one is found. The *getgrnam*( ) function shall return a null
17864 pointer if either the requested entry was not found, or an error occurred. On error, *errno* shall be
17865 set to indicate the error.

17866 The return value may point to a static area which is overwritten by a subsequent call to
17867 *getgrent*( ), *getgrgid*( ), or *getgrnam*( ).

17868 The *getgrnam_r*( ) function shall return zero on success or if the requested entry was not found
17869 and no error has occurred. If any error has occured, an error number shall be returned to
17870 indicate the error.

17871 **ERRORS**

17872 The *getgrnam*( ) and *getgrnam_r*( ) functions may fail if:

17873 [EIO] An I/O error has occurred.

17874 [EINTR] A signal was caught during *getgrnam*( ).

17875 [EMFILE] All file descriptors available to the process are currently open.

17876 [ENFILE] The maximum allowable number of files is currently open in the system.

17877 The *getgrnam_r*( ) function may fail if:

17878 [ERANGE] Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to be
17879 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

endgrent( ), *getgrgid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

Issue 7

SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

Austin Group Interpretation 1003.1-2001 #081 is applied, clarifying the RETURN VALUE section.

The *getgrnam_r*( ) function is moved from the Thread-Safe Functions option to the Base.

NAME

getgroups — get supplementary group IDs

SYNOPSIS

```
#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.

```
#include <sys/types.h>
#include <unistd.h>
...
gid_t *group;
int nogroups;
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.

17960 **RATIONALE**

17961 The related function *setgroups*( ) is a privileged operation and therefore is not covered by this
17962 volume of IEEE Std 1003.1-200x.

17963 As implied by the definition of supplementary groups, the effective group ID may appear in the
17964 array returned by *getgroups*( ) or it may be returned only by *getegid*( ). Duplication may exist, but
17965 the application needs to call *getegid*( ) to be sure of getting all of the information. Various
17966 implementation variations and administrative sequences cause the set of groups appearing in
17967 the result of *getgroups*( ) to vary in order and as to whether the effective group ID is included,
17968 even when the set of groups is the same (in the mathematical sense of "set"). (The history of a
17969 process and its parents could affect the details of the result.)

17970 Application writers should note that {NGROUPS_MAX} is not necessarily a constant on all
17971 implementations.

17972 **FUTURE DIRECTIONS**

17973 None.

17974 **SEE ALSO**

17975 *getegid*( ), *setgid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**,
17976 **<unistd.h>**

17977 **CHANGE HISTORY**

17978 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

17979 **Issue 5**

17980 Normative text previously in the APPLICATION USAGE section is moved to the
17981 DESCRIPTION.

17982 **Issue 6**

17983 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

17984 The following new requirements on POSIX implementations derive from alignment with the
17985 Single UNIX Specification:

17986 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
17987 required for conforming implementations of previous POSIX specifications, it was not
17988 required for UNIX applications.

17989 • A return value of 0 is not permitted, because {NGROUPS_MAX} cannot be 0. This is a FIPS
17990 requirement.

17991 The following changes were made to align with the IEEE P1003.1a draft standard:

17992 • An explanation is added that the effective group ID may be included in the supplementary
17993 group list.

17994 **NAME**

17995 gethostent — network host database functions

17996 **SYNOPSIS**

17997
```
#include <netdb.h>
```

17998
```
struct hostent *gethostent(void);
```

17999 **DESCRIPTION**

18000 Refer to *endhostent( )*.

18001 **NAME**

18002 gethostid — get an identifier for the current host

18003 **SYNOPSIS**

18004 XSI
```
#include <unistd.h>
```
18005
```
long gethostid(void);
```

18006 **DESCRIPTION**

18007 The *gethostid*( ) function shall retrieve a 32-bit identifier for the current host.

18008 **RETURN VALUE**

18009 Upon successful completion, *gethostid*( ) shall return an identifier for the current host.

18010 **ERRORS**

18011 No errors are defined.

18012 **EXAMPLES**

18013 None.

18014 **APPLICATION USAGE**

18015 This volume of IEEE Std 1003.1-200x does not define the domain in which the return value is
18016 unique.

18017 **RATIONALE**

18018 None.

18019 **FUTURE DIRECTIONS**

18020 None.

18021 **SEE ALSO**

18022 *random*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

18023 **CHANGE HISTORY**

18024 First released in Issue 4, Version 2.

18025 **Issue 5**

18026 Moved from X/OPEN UNIX extension to BASE.

18027 **NAME**

18028 gethostname — get name of current host

18029 **SYNOPSIS**

18030
```
#include <unistd.h>
```

18031
```
int gethostname(char *name, size_t namelen);
```

18032 **DESCRIPTION**

18033 The *gethostname*( ) function shall return the standard host name for the current machine. The
18034 *namelen* argument shall specify the size of the array pointed to by the *name* argument. The
18035 returned name shall be null-terminated, except that if *namelen* is an insufficient length to hold
18036 the host name, then the returned name shall be truncated and it is unspecified whether the
18037 returned name is null-terminated.

18038 Host names are limited to {HOST_NAME_MAX} bytes.

18039 **RETURN VALUE**

18040 Upon successful completion, 0 shall be returned; otherwise, –1 shall be returned.

18041 **ERRORS**

18042 No errors are defined.

18043 **EXAMPLES**

18044 None.

18045 **APPLICATION USAGE**

18046 None.

18047 **RATIONALE**

18048 None.

18049 **FUTURE DIRECTIONS**

18050 None.

18051 **SEE ALSO**

18052 *gethostid*( ), *uname*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

18053 **CHANGE HISTORY**

18054 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

18055 The Open Group Base Resolution bwg2001-008 is applied, changing the *namelen* parameter from
18056 **socklen_t** to **size_t**.

NAME

getitimer, setitimer — get and set value of interval timer

SYNOPSIS

OB 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_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.

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.

The interaction between *setitimer*( ) and *alarm*( ) or *sleep*( ) 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.

EXAMPLES

None.

APPLICATION USAGE

Applications should use the *timer_gettime*( ) and *timer_settime*( ) functions instead of the obsolescent *getitimer*( ) and *setitimer*( ) functions, respectively.

RATIONALE

None.

FUTURE DIRECTIONS

The *getitimer*( ) and *setitimer*( ) functions may be removed in a future version.

SEE ALSO

alarm( ), *exec*, *sleep*( ), *timer_getoverrun*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**, **<sys/time.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 *setitimer*( ) prototype for alignment with the ISO/IEC 9899: 1999 standard.

Issue 7

The *getitimer*( ) and *setitimer*( ) functions are marked obsolescent.

18119 **NAME**

18120 getline — read a delimited record from *stream*

18121 **SYNOPSIS**

18122 CX
```
#include <stdio.h>

ssize_t getline(char **lineptr, size_t *n, FILE *stream);
```

18124 **DESCRIPTION**

18125 Refer to *getdelim( )*.

18126 **NAME**

18127 getlogin, getlogin_r — get login name

18128 **SYNOPSIS**

18129
```
#include <unistd.h>
```

18130
```
char *getlogin(void);
```
18131
```
int getlogin_r(char *name, size_t namesize);
```

18132 **DESCRIPTION**

18133 The *getlogin*( ) function shall return a pointer to a string containing the user name associated by
18134 the login activity with the controlling terminal of the current process. If *getlogin*( ) returns a non-
18135 null pointer, then that pointer points to the name that the user logged in under, even if there are
18136 several login names with the same user ID.

18137 The *getlogin*( ) function need not be thread-safe. A function that is not required to be thread-safe
18138 is not required to be reentrant.

18139 The *getlogin_r*( ) function shall put the name associated by the login activity with the controlling
18140 terminal of the current process in the character array pointed to by *name*. The array is *namesize*
18141 characters long and should have space for the name and the terminating null character. The
18142 maximum size of the login name is {LOGIN_NAME_MAX}.

18143 If *getlogin_r*( ) is successful, *name* points to the name the user used at login, even if there are
18144 several login names with the same user ID.

18145 **RETURN VALUE**

18146 Upon successful completion, *getlogin*( ) shall return a pointer to the login name or a null pointer
18147 if the user's login name cannot be found. Otherwise, it shall return a null pointer and set *errno* to
18148 indicate the error.

18149 The return value from *getlogin*( ) may point to static data whose content is overwritten by each
18150 call.

18151 If successful, the *getlogin_r*( ) function shall return zero; otherwise, an error number shall be
18152 returned to indicate the error.

18153 **ERRORS**

18154 The *getlogin*( ) and *getlogin_r*( ) functions may fail if:

18155 [EMFILE] All file descriptors available to the process are currently open.

18156 [ENFILE] The maximum allowable number of files is currently open in the system.

18157 [ENXIO] The calling process has no controlling terminal.

18158 The *getlogin_r*( ) function may fail if:

18159 [ERANGE] The value of *namesize* is smaller than the length of the string to be returned
18160 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.

```
#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:

```
getpwuid(geteuid())
```

while the information provided by historical implementations of *cuserid*( ) can be obtained by:

```
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 1003.1-200x, **<limits.h>**, **<unistd.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 *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.

Issue 7

SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

The *getlogin_r*( ) function is moved from the Thread-Safe Functions option to the Base.

18223 **NAME**

18224 getmsg, getpmsg — receive next message from a STREAMS file (**STREAMS**)

18225 **SYNOPSIS**

18226 OB XSR
```
#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);
```

18232 **DESCRIPTION**

18233 The *getmsg*( ) function shall retrieve the contents of a message located at the head of the
18234 STREAM head read queue associated with a STREAMS file and place the contents into one or
18235 more buffers. The message contains either a data part, a control part, or both. The data and
18236 control parts of the message shall be placed into separate buffers, as described below. The
18237 semantics of each part are defined by the originator of the message.

18238 The *getpmsg*( ) function shall be equivalent to *getmsg*( ), except that it provides finer control over
18239 the priority of the messages received. Except where noted, all requirements on *getmsg*( ) also
18240 pertain to *getpmsg*( ).

18241 The *fildes* argument specifies a file descriptor referencing a STREAMS-based file.

18242 The *ctlptr* and *dataptr* arguments each point to a **strbuf** structure, in which the *buf* member points
18243 to a buffer in which the data or control information is to be placed, and the *maxlen* member
18244 indicates the maximum number of bytes this buffer can hold. On return, the *len* member shall
18245 contain the number of bytes of data or control information actually received. The *len* member
18246 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
18247 control information is present in the message.

18248 When *getmsg*( ) is called, *flagsp* should point to an integer that indicates the type of message the
18249 process is able to receive. This is described further below.

18250 The *ctlptr* argument is used to hold the control part of the message, and *dataptr* is used to hold
18251 the data part of the message. If *ctlptr* (or *dataptr*) is a null pointer or the *maxlen* member is −1, the
18252 control (or data) part of the message shall not be processed and shall be left on the STREAM
18253 head read queue, and if the *ctlptr* (or *dataptr*) is not a null pointer, *len* shall be set to −1. If the
18254 *maxlen* member is set to 0 and there is a zero-length control (or data) part, that zero-length part
18255 shall be removed from the read queue and *len* shall be set to 0. If the *maxlen* member is set to 0
18256 and there are more than 0 bytes of control (or data) information, that information shall be left on
18257 the read queue and *len* shall be set to 0. If the *maxlen* member in *ctlptr* (or *dataptr*) is less than the
18258 control (or data) part of the message, *maxlen* bytes shall be retrieved. In this case, the remainder
18259 of the message shall be left on the STREAM head read queue and a non-zero return value shall
18260 be provided.

18261 By default, *getmsg*( ) shall process the first available message on the STREAM head read queue.
18262 However, a process may choose to retrieve only high-priority messages by setting the integer
18263 pointed to by *flagsp* to RS_HIPRI. In this case, *getmsg*( ) shall only process the next message if it is
18264 a high-priority message. When the integer pointed to by *flagsp* is 0, any available message shall
18265 be retrieved. In this case, on return, the integer pointed to by *flagsp* shall be set to RS_HIPRI if a
18266 high-priority message was retrieved, or 0 otherwise.

18267 For *getpmsg*( ), the flags are different. The *flagsp* argument points to a bitmask with the following
18268 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.

[ENOSTR] 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 STREAM-head read queue, returning control and data information to the buffers pointed to by *ctrlbuf* and *databuf*, respectively.

```
#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 STREAM-head read queue.

```
#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);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

The *getmsg*( ) and *getpmsg*( ) functions may be removed in a future version.

SEE ALSO

Section 2.6 (on page 38), *poll*( ), *putmsg*( ), *read*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stropts.h>**

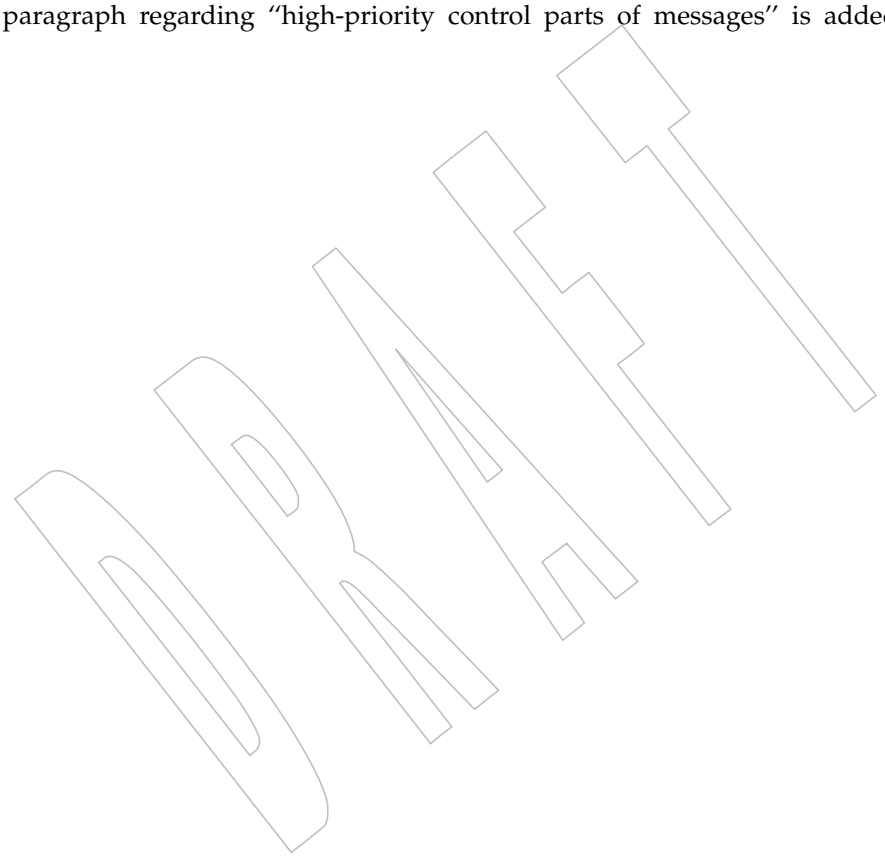
CHANGE HISTORY

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



18378 **NAME**

18379 getnameinfo — get name information

18380 **SYNOPSIS**

18381 ```
#include <sys/socket.h>
```
18382 ```
#include <netdb.h>
```

18383 ```
int getnameinfo(const struct sockaddr *restrict sa, socklen_t salen,
```
18384 ```
    char *restrict node, socklen_t nodelen, char *restrict service,
```
18385 ```
    socklen_t servicelen, int flags);
```

18386 **DESCRIPTION**

18387 The *getnameinfo*( ) function shall translate a socket address to a node name and service location,
18388 all of which are defined as in *getaddrinfo*( ).

18389 The *sa* argument points to a socket address structure to be translated.

18390 IP6 If the socket address structure contains an IPv4-mapped IPv6 address or an IPv4-compatible
18391 IPv6 address, the implementation shall extract the embedded IPv4 address and lookup the node
18392 name for that IPv4 address.

18393 If the address is the IPv6 unspecified address ("::"), a lookup shall not be performed and the
18394 behavior shall be the same as when the node's name cannot be located.

18395 If the *node* argument is non-NULL and the *nodelen* argument is non-zero, then the *node* argument
18396 points to a buffer able to contain up to *nodelen* characters that receives the node name as a null-
18397 terminated string. If the *node* argument is NULL or the *nodelen* argument is zero, the node name
18398 shall not be returned. If the node's name cannot be located, the numeric form of the address
18399 contained in the socket address structure pointed to by the *sa* argument is returned instead of its
18400 name.

18401 If the *service* argument is non-NULL and the *servicelen* argument is non-zero, then the *service*
18402 argument points to a buffer able to contain up to *servicelen* bytes that receives the service name
18403 as a null-terminated string. If the *service* argument is NULL or the *servicelen* argument is zero,
18404 the service name shall not be returned. If the service's name cannot be located, the numeric form
18405 of the service address (for example, its port number) shall be returned instead of its name.

18406 The *flags* argument is a flag that changes the default actions of the function. By default the fully-
18407 qualified domain name (FQDN) for the host shall be returned, but:

18408 • If the flag bit NI_NOFQDN is set, only the node name portion of the FQDN shall be
18409 returned for local hosts.

18410 • If the flag bit NI_NUMERICHOST is set, the numeric form of the address contained in the
18411 socket address structure pointed to by the *sa* argument shall be returned instead of its
18412 name.

18413 • If the flag bit NI_NAMEREQD is set, an error shall be returned if the host's name cannot
18414 be located.

18415 • If the flag bit NI_NUMERICSERV is set, the numeric form of the service address shall be
18416 returned (for example, its port number) instead of its name.

18417 • If the flag bit NI_NUMERICSCOPE is set, the numeric form of the scope identifier shall be
18418 returned (for example, interface index) instead of its name. This flag shall be ignored if the
18419 *sa* argument is not an IPv6 address.
```

- 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.
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_OVERFLOW] An argument buffer overflowed. The buffer pointed to by the *node* argument or the *service* argument was too small.

[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"`.

The IPv6 unspecified address (`"::"`) and the IPv6 loopback address (`"::1"`) are not IPv4-compatible addresses.

18460 **RATIONALE**

18461 None.

18462 **FUTURE DIRECTIONS**

18463 None.

18464 **SEE ALSO**

18465 *gai_strerror*( ), *getaddrinfo*( ), *getservbyname*( ), *inet_ntop*( ), *socket*( ), the Base Definitions volume of
18466 IEEE Std 1003.1-200x, **<netdb.h>**, **<sys/socket.h>**

18467 **CHANGE HISTORY**

18468 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

18469 The **restrict** keyword is added to the *getnameinfo*( ) prototype for alignment with the
18470 ISO/IEC 9899:1999 standard.

18471 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/23 is applied, making various changes in
18472 the SYNOPSIS and DESCRIPTION for alignment with IPv6.

18473 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/24 is applied, adding the
18474 [EAI_OVERFLOW] error to the ERRORS section.

18475 **Issue 7**

18476 SD5-XSH-ERN-127 is applied, clarifying the behavior if the address is the IPv6 unspecified
18477 address.

18478 **NAME**

18479 getnetbyaddr, getnetbyname, getnetent — network database functions

18480 **SYNOPSIS**

18481
```
#include <netdb.h>
```

18482
```
struct netent *getnetbyaddr(uint32_t net, int type);
```
18483
```
struct netent *getnetbyname(const char *name);
```
18484
```
struct netent *getnetent(void);
```

18485 **DESCRIPTION**

18486 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 1003.1-200x, 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 thread-safe. A function that is not required to be thread-safe is

not required to be reentrant.

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:

```
#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;
            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 –oarg path path
cmd –aoarg path path
```

Checking Options and Arguments

The following example parses a set of command line options and prints messages to standard output for each option and argument that it encounters.

```
#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

None.

18675 **SEE ALSO**

18676 *exec*, the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**, the Shell and Utilities
18677 volume of IEEE Std 1003.1-200x

18678 **CHANGE HISTORY**

18679 First released in Issue 1. Derived from Issue 1 of the SVID.

18680 **Issue 5**

18681 A note indicating that the *getopt*( ) function need not be reentrant is added to the DESCRIPTION.

18682 **Issue 6**

18683 IEEE PASC Interpretation 1003.2 #150 is applied.

18684 **NAME**

18685 getpeername — get the name of the peer socket

18686 **SYNOPSIS**

18687
```
#include <sys/socket.h>
```

18688
```
int getpeername(int socket, struct sockaddr *restrict address,
    socklen_t *restrict address_len);
```

18690 **DESCRIPTION**

18691 The *getpeername*( ) function shall retrieve the peer address of the specified socket, store this
18692 address in the **sockaddr** structure pointed to by the *address* argument, and store the length of this
18693 address in the object pointed to by the *address_len* argument.

18694 If the actual length of the address is greater than the length of the supplied **sockaddr** structure,
18695 the stored address shall be truncated.

18696 If the protocol permits connections by unbound clients, and the peer is not bound, then the
18697 value stored in the object pointed to by *address* is unspecified.

18698 **RETURN VALUE**

18699 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to
18700 indicate the error.

18701 **ERRORS**

18702 The *getpeername*( ) function shall fail if:

18703 [EBADF] The *socket* argument is not a valid file descriptor.

18704 [EINVAL] The socket has been shut down.

18705 [ENOTCONN] The socket is not connected or otherwise has not had the peer pre-specified.

18706 [ENOTSOCK] The *socket* argument does not refer to a socket.

18707 [EOPNOTSUPP] The operation is not supported for the socket protocol.

18708 The *getpeername*( ) function may fail if:

18709 [ENOBUFS] Insufficient resources were available in the system to complete the call.

18710 **EXAMPLES**

18711 None.

18712 **APPLICATION USAGE**

18713 None.

18714 **RATIONALE**

18715 None.

18716 **FUTURE DIRECTIONS**

18717 None.

18718 **SEE ALSO**

18719 *accept*( ), *bind*( ), *getsockname*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
18720 **<sys/socket.h>**

18721 **CHANGE HISTORY**

18722 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

18723 The **restrict** keyword is added to the *getpeername*( ) prototype for alignment with the
18724 ISO/IEC 9899: 1999 standard.

18725 **NAME**

18726 getpgid — get the process group ID for a process

18727 **SYNOPSIS**

18728 `#include <unistd.h>`

18729 `pid_t getpgid(pid_t pid);`

18730 **DESCRIPTION**

18731 The *getpgid*( ) function shall return the process group ID of the process whose process ID is equal
18732 to *pid*. If *pid* is equal to 0, *getpgid*( ) shall return the process group ID of the calling process.

18733 **RETURN VALUE**

18734 Upon successful completion, *getpgid*( ) shall return a process group ID. Otherwise, it shall return
18735 (**pid_t**)–1 and set *errno* to indicate the error.

18736 **ERRORS**

18737 The *getpgid*( ) function shall fail if:

18738 [EPERM] The process whose process ID is equal to *pid* is not in the same session as the
18739 calling process, and the implementation does not allow access to the process
18740 group ID of that process from the calling process.

18741 [ESRCH] There is no process with a process ID equal to *pid*.

18742 The *getpgid*( ) function may fail if:

18743 [EINVAL] The value of the *pid* argument is invalid.

18744 **EXAMPLES**

18745 None.

18746 **APPLICATION USAGE**

18747 None.

18748 **RATIONALE**

18749 None.

18750 **FUTURE DIRECTIONS**

18751 None.

18752 **SEE ALSO**

18753 *exec*, *fork*( ), *getpgrp*( ), *getpid*( ), *getsid*( ), *setpgid*( ), *setsid*( ), the Base Definitions volume of
18754 IEEE Std 1003.1-200x, **<unistd.h>**

18755 **CHANGE HISTORY**

18756 First released in Issue 4, Version 2.

18757 **Issue 5**

18758 Moved from X/OPEN UNIX extension to BASE.

18759 **Issue 7**

18760 The *getpgid*( ) function is moved from the XSI option to the Base.

NAME

getpgrp — get the process group ID of the calling process

SYNOPSIS

```
#include <unistd.h>

pid_t getpgrp(void);
```

DESCRIPTION

The *getpgrp*( ) function shall return the process group ID of the calling process.

RETURN VALUE

The *getpgrp*( ) 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 *getpgrp*( ) 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 *getpgrp*( ) suffices, and the added complexity of the 4.3 BSD *getpgrp*( ) is provided by the XSI extension *getpgid*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *fork*( ), *getpgid*( ), *getpid*( ), *getppid*( ), *kill*( ), *setpgid*( ), *setsid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

18796 **NAME**

18797 getpid — get the process ID

18798 **SYNOPSIS**

18799
```
#include <unistd.h>
```

18800
```
pid_t getpid(void);
```

18801 **DESCRIPTION**

18802 The *getpid*( ) function shall return the process ID of the calling process.

18803 **RETURN VALUE**

18804 The *getpid*( ) function shall always be successful and no return value is reserved to indicate an
18805 error.

18806 **ERRORS**

18807 No errors are defined.

18808 **EXAMPLES**

18809 None.

18810 **APPLICATION USAGE**

18811 None.

18812 **RATIONALE**

18813 None.

18814 **FUTURE DIRECTIONS**

18815 None.

18816 **SEE ALSO**

18817 *exec*, *fork*( ), *getpgrp*( ), *getppid*( ), *kill*( ), *mkdtemp*( ), *setpgid*( ), *setsid*( ), the Base Definitions volume
18818 of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

18819 **CHANGE HISTORY**

18820 First released in Issue 1. Derived from Issue 1 of the SVID.

18821 **Issue 6**

18822 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

18823 The following new requirements on POSIX implementations derive from alignment with the
18824 Single UNIX Specification:

18825 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
18826 required for conforming implementations of previous POSIX specifications, it was not
18827 required for UNIX applications.

18828 **NAME**

18829 getpmsg — receive next message from a STREAMS file

18830 **SYNOPSIS**

18831 OB XSI
```
#include <stropts.h>

int getpmsg(int fildes, struct strbuf *restrict ctlptr,
    struct strbuf *restrict dataptr, int *restrict bandp,
    int *restrict flagsp);
```

18835 **DESCRIPTION**

18836 Refer to *getmsg( )*.

18837 **NAME**

18838 getppid — get the parent process ID

18839 **SYNOPSIS**

18840
```
#include <unistd.h>
```

18841
```
pid_t getppid(void);
```

18842 **DESCRIPTION**

18843 The *getppid*( ) function shall return the parent process ID of the calling process.

18844 **RETURN VALUE**

18845 The *getppid*( ) function shall always be successful and no return value is reserved to indicate an
18846 error.

18847 **ERRORS**

18848 No errors are defined.

18849 **EXAMPLES**

18850 None.

18851 **APPLICATION USAGE**

18852 None.

18853 **RATIONALE**

18854 None.

18855 **FUTURE DIRECTIONS**

18856 None.

18857 **SEE ALSO**

18858 *exec*, *fork*( ), *getpgid*( ), *getpgrp*( ), *getpid*( ), *kill*( ), *setpgid*( ), *setsid*( ), the Base Definitions volume of
18859 IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

18860 **CHANGE HISTORY**

18861 First released in Issue 1. Derived from Issue 1 of the SVID.

18862 **Issue 6**

18863 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

18864 The following new requirements on POSIX implementations derive from alignment with the
18865 Single UNIX Specification:

18866 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
18867 required for conforming implementations of previous POSIX specifications, it was not
18868 required for UNIX applications.

NAME

getpriority, setpriority — get and set the nice value

SYNOPSIS

XSI

```
#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 {NZERO} 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.

PS | TPS

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.

RETURN 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.

18909 **ERRORS**

18910 The *getpriority*( ) and *setpriority*( ) functions shall fail if:

18911 [ESRCH] No process could be located using the *which* and *who* argument values
18912 specified.

18913 [EINVAL] The value of the *which* argument was not recognized, or the value of the *who*
18914 argument is not a valid process ID, process group ID, or user ID.

18915 In addition, *setpriority*( ) may fail if:

18916 [EPERM] A process was located, but neither the real nor effective user ID of the
18917 executing process match the effective user ID of the process whose nice value
18918 is being changed.

18919 [EACCES] A request was made to change the nice value to a lower numeric value and the
18920 current process does not have appropriate privileges.

18921 **EXAMPLES**

18922 **Using getpriority( )**

18923 The following example returns the current scheduling priority for the process ID returned by the
18924 call to *getpid*( ).

```
18925 #include <sys/resource.h>
18926 ...
18927 int which = PRIO_PROCESS;
18928 id_t pid;
18929 int ret;

18930 pid = getpid();
18931 ret = getpriority(which, pid);
```

18932 **Using setpriority( )**

18933 The following example sets the priority for the current process ID to –20.

```
18934 #include <sys/resource.h>
18935 ...
18936 int which = PRIO_PROCESS;
18937 id_t pid;
18938 int priority = -20;
18939 int ret;

18940 pid = getpid();
18941 ret = setpriority(which, pid, priority);
```

18942 **APPLICATION USAGE**

18943 The *getpriority*( ) and *setpriority*( ) functions work with an offset nice value (nice value
18944 –{NZERO}). The nice value is in the range [0,2*{NZERO} –1], while the return value for
18945 *getpriority*( ) and the third parameter for *setpriority*( ) are in the range [–{NZERO},{NZERO} –1].

18946 **RATIONALE**

18947 None.

18948 **FUTURE DIRECTIONS**

18949 None.

18950 **SEE ALSO**

18951 *nice( )*, *sched_get_priority_max( )*, *sched_setscheduler( )*, the Base Definitions volume of
18952 IEEE Std 1003.1-200x, **<sys/resource.h>**

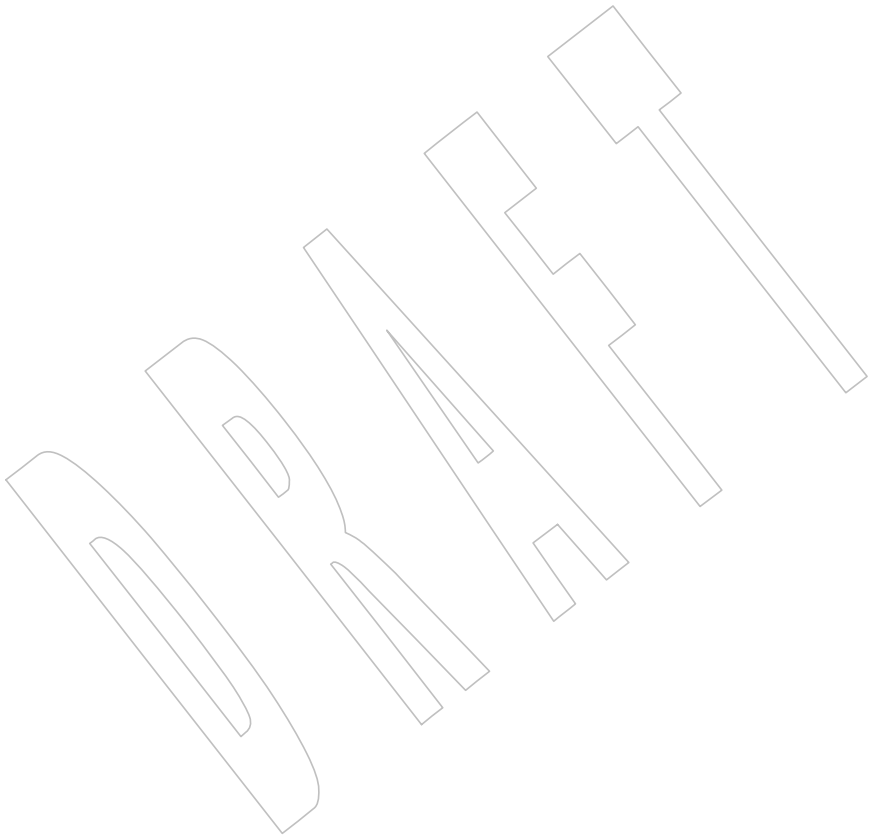
18953 **CHANGE HISTORY**

18954 First released in Issue 4, Version 2.

18955 **Issue 5**

18956 Moved from X/OPEN UNIX extension to BASE.

18957 The DESCRIPTION is reworded in terms of the nice value rather than *priority* to avoid confusion
18958 with functionality in the POSIX Realtime Extension.



18959 **NAME**

18960 getprotobyname, getprotobynumber, getprotent — network protocol database functions

18961 **SYNOPSIS**

18962
```
#include <netdb.h>
```

18963
```
struct protoent *getprotobyname(const char *name);
```
18964
```
struct protoent *getprotobynumber(int proto);
```
18965
```
struct protoent *getprotoent(void);
```

18966 **DESCRIPTION**

18967 Refer to *endprotoent( )*.

18968 **NAME**

18969 getpwent — get user database entry

18970 **SYNOPSIS**

18971 XSI
```
#include <pwd.h>

struct passwd *getpwent(void);
```
18972

18973 **DESCRIPTION**

18974 Refer to *endpwent( )*.

18975 **NAME**

18976 getpwnam, getpwnam_r — search user database for a name

18977 **SYNOPSIS**

18978
```
#include <pwd.h>
```

18979
```
struct passwd *getpwnam(const char *name);
```
18980
```
int getpwnam_r(const char *name, struct passwd *pwd, char *buffer,
```
18981
```
    size_t bufsize, struct passwd **result);
```

18982 **DESCRIPTION**

18983 The *getpwnam*( ) function shall search the user database for an entry with a matching *name*.

18984 The *getpwnam*( ) function need not be thread-safe. A function that is not required to be thread-
18985 safe is not required to be reentrant.

18986 Applications wishing to check for error situations should set *errno* to 0 before calling
18987 *getpwnam*( ). If *getpwnam*( ) returns a null pointer and *errno* is non-zero, an error occurred.

18988 The *getpwnam_r*( ) function shall update the **passwd** structure pointed to by *pwd* and store a
18989 pointer to that structure at the location pointed to by *result*. The structure shall contain an entry
18990 from the user database with a matching *name*. Storage referenced by the structure is allocated
18991 from the memory provided with the *buffer* parameter, which is *bufsize* bytes in size. The
18992 maximum size needed for this buffer can be determined with the {_SC_GETPW_R_SIZE_MAX}
18993 *sysconf*( ) parameter. A NULL pointer shall be returned at the location pointed to by *result* on
18994 error or if the requested entry is not found.

18995 **RETURN VALUE**

18996 The *getpwnam*( ) function shall return a pointer to a **struct passwd** with the structure as defined
18997 in **<pwd.h>** with a matching entry if found. A null pointer shall be returned if the requested
18998 entry is not found, or an error occurs. On error, *errno* shall be set to indicate the error.

18999 The return value may point to a static area which is overwritten by a subsequent call to
19000 *getpwent*( ), *getpwnam*( ), or *getpwuid*( ).

19001 The *getpwnam_r*( ) function shall return zero on success or if the requested entry was not found
19002 and no error has occurred. If an error has occurred, an error number shall be returned to indicate
19003 the error.

19004 **ERRORS**

19005 The *getpwnam*( ) and *getpwnam_r*( ) functions may fail if:

19006 [EIO] An I/O error has occurred.

19007 [EINTR] A signal was caught during *getpwnam*( ).

19008 [EMFILE] All file descriptors available to the process are currently open.

19009 [ENFILE] The maximum allowable number of files is currently open in the system.

19010 The *getpwnam_r*( ) function may fail if:

19011 [ERANGE] Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to be
19012 referenced by the resulting **passwd** structure.

EXAMPLES

Getting an Entry for the Login Name

The following example uses the *getlogin*( ) function to return the name of the user who logged in; this information is passed to the *getpwnam*( ) function to get the user database entry for that user.

```
#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: *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 *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

getpwuid( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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 *getpwnam_r*( ) function is included for alignment with the POSIX Threads Extension.

A note indicating that the *getpwnam*( ) function need not be reentrant is added to the DESCRIPTION.

Issue 6

The *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 *name*.

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 [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.

Issue 7

SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

The *getpwnam_r*( ) function is moved from the Thread-Safe Functions option to the Base.

19072 **NAME**

19073 getpwuid, getpwuid_r — search user database for a user ID

19074 **SYNOPSIS**

19075
```
#include <pwd.h>
```

19076
```
struct passwd *getpwuid(uid_t uid);
```
19077
```
int getpwuid_r(uid_t uid, struct passwd *pwd, char *buffer,
```
19078
```
    size_t bufsize, struct passwd **result);
```

19079 **DESCRIPTION**

19080 The *getpwuid*( ) function shall search the user database for an entry with a matching *uid*.

19081 The *getpwuid*( ) function need not be thread-safe. A function that is not required to be thread-safe
19082 is not required to be reentrant.

19083 Applications wishing to check for error situations should set *errno* to 0 before calling *getpwuid*( ).
19084 If *getpwuid*( ) returns a null pointer and *errno* is set to non-zero, an error occurred.

19085 The *getpwuid_r*( ) function shall update the **passwd** structure pointed to by *pwd* and store a
19086 pointer to that structure at the location pointed to by *result*. The structure shall contain an entry
19087 from the user database with a matching *uid*. Storage referenced by the structure is allocated
19088 from the memory provided with the *buffer* parameter, which is *bufsize* bytes in size. The
19089 maximum size needed for this buffer can be determined with the {_SC_GETPW_R_SIZE_MAX}
19090 *sysconf*( ) parameter. A NULL pointer shall be returned at the location pointed to by *result* on
19091 error or if the requested entry is not found.

19092 **RETURN VALUE**

19093 The *getpwuid*( ) function shall return a pointer to a **struct passwd** with the structure as defined in
19094 **<pwd.h>** with a matching entry if found. A null pointer shall be returned if the requested entry
19095 is not found, or an error occurs. On error, *errno* shall be set to indicate the error.

19096 The return value may point to a static area which is overwritten by a subsequent call to
19097 *getpwent*( ), *getpwnam*( ), or *getpwuid*( ).

19098 If successful, the *getpwuid_r*( ) function shall return zero; otherwise, an error number shall be
19099 returned to indicate the error.

19100 **ERRORS**

19101 The *getpwuid*( ) and *getpwuid_r*( ) functions may fail if:

19102 [EIO] An I/O error has occurred.

19103 [EINTR] A signal was caught during *getpwuid*( ).

19104 [EMFILE] All file descriptors available to the process are currently open.

19105 [ENFILE] The maximum allowable number of files is currently open in the system.

19106 The *getpwuid_r*( ) function may fail if:

19107 [ERANGE] Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to be
19108 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).

```
#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.

```
#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.

```
#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.

19152 **RATIONALE**
19153 None.

19154 **FUTURE DIRECTIONS**
19155 None.

19156 **SEE ALSO**
19157 *getpwnam*( ), *geteuid*( ), *getuid*( ), *getlogin*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
19158 **<limits.h>**, **<pwd.h>**, **<sys/types.h>**

19159 **CHANGE HISTORY**
19160 First released in Issue 1. Derived from System V Release 2.0.

19161 **Issue 5**
19162 Normative text previously in the APPLICATION USAGE section is moved to the RETURN
19163 VALUE section.

19164 The *getpwuid_r*( ) function is included for alignment with the POSIX Threads Extension.

19165 A note indicating that the *getpwuid*( ) function need not be reentrant is added to the
19166 DESCRIPTION.

19167 **Issue 6**
19168 The *getpwuid_r*( ) function is marked as part of the Thread-Safe Functions option.

19169 The Open Group Corrigendum U028/3 is applied, correcting text in the DESCRIPTION
19170 describing matching the *uid*.

19171 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

19172 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

19173 The following new requirements on POSIX implementations derive from alignment with the
19174 Single UNIX Specification:

19175 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
19176 required for conforming implementations of previous POSIX specifications, it was not
19177 required for UNIX applications.

19178 • In the RETURN VALUE section, the requirement to set *errno* on error is added.

19179 • The [EIO], [EINTR], [EMFILE], and [ENFILE] optional error conditions are added.

19180 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
19181 its avoidance of possibly using a static data area.

19182 IEEE PASC Interpretation 1003.1 #116 is applied, changing the description of the size of the
19183 buffer from *bufsize* characters to bytes.

19184 **Issue 7**
19185 SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

19186 The *getpwuid_r*( ) function is moved from the Thread-Safe Functions option to the Base.

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 data segment of the process, 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 a file descriptor shall 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 the initial thread's 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 total available memory of the process, 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.

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.

If a process attempts to set the hard limit or soft limit for RLIMIT_NOFILE to less than the highest currently open file descriptor +1, unexpected behavior may occur.

RATIONALE

It should be noted that RLIMIT_STACK applies "at least" to the stack of the initial thread in the process, and not to the sum of all the stacks in the process, as that would be very limiting unless the value is so big as to provide no value at all with a single thread.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *fork( )*, *malloc( )*, *open( )*, *sigaltstack( )*, *sysconf( )*, *ulimit( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

Issue 6

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/25 is applied, changing wording for RLIMIT_NOFILE in the DESCRIPTION related to functions that allocate a file descriptor failing with [EMFILE]. Text is added to the APPLICATION USAGE section noting the consequences of a process attempting to set the hard or soft limit for RLIMIT_NOFILE less than the highest currently open file descriptor +1.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/46 is applied, updating the definition of RLIMIT_STACK in the DESCRIPTION section from "the maximum size of a process stack" to "the maximum size of the initial thread's stack". Text is added to the RATIONALE section.

19301 **NAME**

19302 getrusage — get information about resource utilization

19303 **SYNOPSIS**

19304 XSI
```
#include <sys/resource.h>
```
19305
```
int getrusage(int who, struct rusage *r_usage);
```

19306 **DESCRIPTION**

19307 The *getrusage*( ) function shall provide measures of the resources used by the current process or
19308 its terminated and waited-for child processes. If the value of the *who* argument is
19309 RUSAGE_SELF, information shall be returned about resources used by the current process. If the
19310 value of the *who* argument is RUSAGE_CHILDREN, information shall be returned about
19311 resources used by the terminated and waited-for children of the current process. If the child is
19312 never waited for (for example, if the parent has SA_NOCLDWAIT set or sets SIGCHLD to
19313 SIG_IGN), the resource information for the child process is discarded and not included in the
19314 resource information provided by *getrusage*( ).

19315 The *r_usage* argument is a pointer to an object of type **struct rusage** in which the returned
19316 information is stored.

19317 **RETURN VALUE**

19318 Upon successful completion, *getrusage*( ) shall return 0; otherwise, –1 shall be returned and *errno*
19319 set to indicate the error.

19320 **ERRORS**

19321 The *getrusage*( ) function shall fail if:

19322 [EINVAL] The value of the *who* argument is not valid.

19323 **EXAMPLES**

19324 **Using getrusage( )**

19325 The following example returns information about the resources used by the current process.

19326
```
#include <sys/resource.h>
```
19327
```
...
```
19328
```
int who = RUSAGE_SELF;
```
19329
```
struct rusage usage;
int ret;
```
19330
19331
```
ret = getrusage(who, &usage);
```

19332 **APPLICATION USAGE**

19333 None.

19334 **RATIONALE**

19335 None.

19336 **FUTURE DIRECTIONS**

19337 None.

19338 **SEE ALSO**

19339 *exit*( ), *sigaction*( ), *time*( ), *times*( ), *wait*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
19340 **<sys/resource.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

19345 **NAME**

19346 gets — get a string from a stdin stream

19347 **SYNOPSIS**

19348
```
#include <stdio.h>
```

19349
```
char *gets(char *s);
```

19350 **DESCRIPTION**

19351 CX The functionality described on this reference page is aligned with the ISO C standard. Any
19352 conflict between the requirements described here and the ISO C standard is unintentional. This
19353 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

19354 The *gets*( ) function shall read bytes from the standard input stream, *stdin*, into the array pointed
19355 to by *s*, until a <newline> is read or an end-of-file condition is encountered. Any <newline> shall
19356 be discarded and a null byte shall be placed immediately after the last byte read into the array.

19357 CX The *gets*( ) function may mark the *st_atime* field of the file associated with *stream* for update. The
19358 *st_atime* field shall be marked for update by the first successful execution of *fgetc*( ), *fgets*( ),
19359 *fread*( ), *getc*( ), *getchar*( ), *gets*( ), *fscanf*( ), or *scanf*( ) using *stream* that returns data not supplied by
19360 a prior call to *ungetc*( ).

19361 **RETURN VALUE**

19362 Upon successful completion, *gets*( ) shall return *s*. If the end-of-file indicator for the stream is set,
19363 or if the stream is at end-of-file, the end-of-file indicator for the stream shall be set and *gets*( )
19364 shall return a null pointer. If a read error occurs, the error indicator for the stream shall be set,
19365 CX *gets*( ) shall return a null pointer, and set *errno* to indicate the error.

19366 **ERRORS**

19367 Refer to *fgetc*( ).

19368 **EXAMPLES**

19369 None.

19370 **APPLICATION USAGE**

19371 Reading a line that overflows the array pointed to by *s* results in undefined behavior. The use of
19372 *fgets*( ) is recommended.

19373 Since the user cannot specify the length of the buffer passed to *gets*( ), use of this function is
19374 discouraged. The length of the string read is unlimited. It is possible to overflow this buffer in
19375 such a way as to cause applications to fail, or possible system security violations.

19376 It is recommended that the *fgets*( ) function should be used to read input lines.

19377 **RATIONALE**

19378 None.

19379 **FUTURE DIRECTIONS**

19380 None.

19381 **SEE ALSO**

19382 *feof*( ), *ferror*( ), *fgets*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

19383 **CHANGE HISTORY**

19384 First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are marked.

Issue 7

Austin Group Interpretation 1003.1-2001 #051 is applied, clarifying the RETURN VALUE section.

19389 **NAME**

19390 getservbyname, getservbyport, getservent — network services database functions

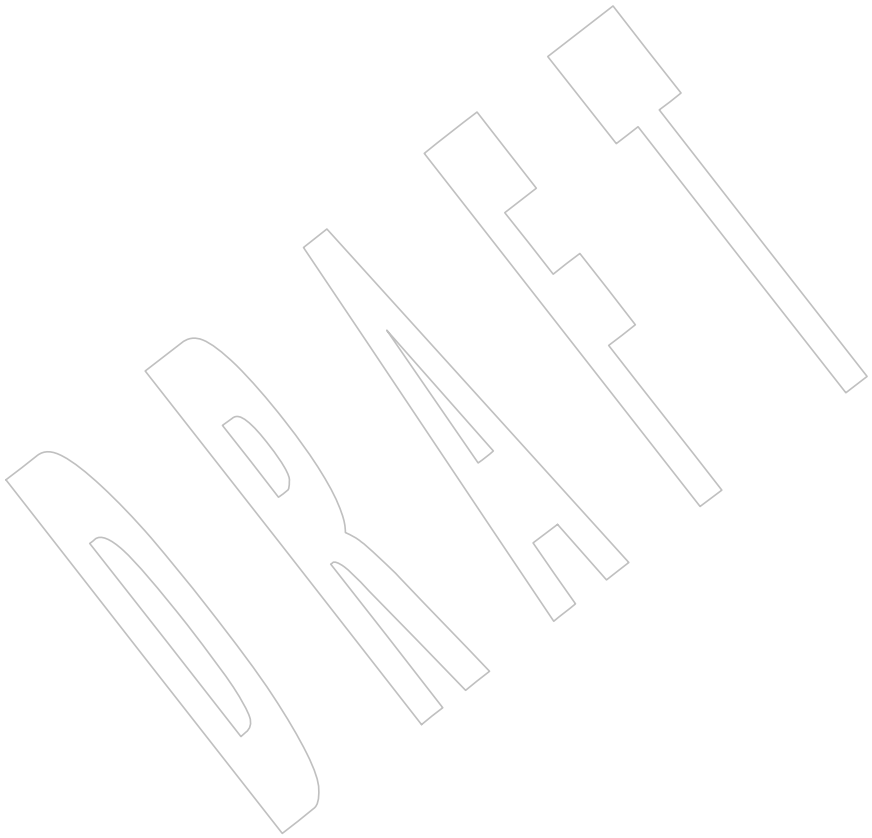
19391 **SYNOPSIS**

19392
```
#include <netdb.h>
```

19393
```
struct servent *getservbyname(const char *name, const char *proto);
```
19394
```
struct servent *getservbyport(int port, const char *proto);
```
19395
```
struct servent *getservent(void);
```

19396 **DESCRIPTION**

19397 Refer to *endservent( )*.



19398 **NAME**

19399 getsid — get the process group ID of a session leader

19400 **SYNOPSIS**

19401 ```
#include <unistd.h>
```

19402 ```
pid_t getsid(pid_t pid);
```

19403 **DESCRIPTION**

19404 The *getsid*( ) function shall obtain the process group ID of the process that is the session leader of
19405 the process specified by *pid*. If *pid* is (**pid_t**)0, it specifies the calling process.

19406 **RETURN VALUE**

19407 Upon successful completion, *getsid*( ) shall return the process group ID of the session leader of
19408 the specified process. Otherwise, it shall return (**pid_t**)–1 and set *errno* to indicate the error.

19409 **ERRORS**

19410 The *getsid*( ) function shall fail if:

19411 [EPERM] The process specified by *pid* is not in the same session as the calling process,
19412 and the implementation does not allow access to the process group ID of the
19413 session leader of that process from the calling process.

19414 [ESRCH] There is no process with a process ID equal to *pid*.

19415 **EXAMPLES**

19416 None.

19417 **APPLICATION USAGE**

19418 None.

19419 **RATIONALE**

19420 None.

19421 **FUTURE DIRECTIONS**

19422 None.

19423 **SEE ALSO**

19424 *exec*, *fork*( ), *getpid*( ), *getpgid*( ), *setpgid*( ), *setsid*( ), the Base Definitions volume of
19425 IEEE Std 1003.1-200x, **<unistd.h>**

19426 **CHANGE HISTORY**

19427 First released in Issue 4, Version 2.

19428 **Issue 5**

19429 Moved from X/OPEN UNIX extension to BASE.

19430 **Issue 7**

19431 The *getsid*( ) function is moved from the XSI option to the Base.

19432 **NAME**

19433 getsockname — get the socket name

19434 **SYNOPSIS**

19435
```
#include <sys/socket.h>
```

19436
```
int getsockname(int socket, struct sockaddr *restrict address,
    socklen_t *restrict address_len);
```

19438 **DESCRIPTION**

19439 The *getsockname*( ) function shall retrieve the locally-bound name of the specified socket, store
19440 this address in the **sockaddr** structure pointed to by the *address* argument, and store the length of
19441 this address in the object pointed to by the *address_len* argument.

19442 If the actual length of the address is greater than the length of the supplied **sockaddr** structure,
19443 the stored address shall be truncated.

19444 If the socket has not been bound to a local name, the value stored in the object pointed to by
19445 *address* is unspecified.

19446 **RETURN VALUE**

19447 Upon successful completion, 0 shall be returned, the *address* argument shall point to the address
19448 of the socket, and the *address_len* argument shall point to the length of the address. Otherwise, –1
19449 shall be returned and *errno* set to indicate the error.

19450 **ERRORS**

19451 The *getsockname*( ) function shall fail if:

19452 [EBADF] The *socket* argument is not a valid file descriptor.

19453 [ENOTSOCK] The *socket* argument does not refer to a socket.

19454 [EOPNOTSUPP] The operation is not supported for this socket's protocol.

19455 The *getsockname*( ) function may fail if:

19456 [EINVAL] The socket has been shut down.

19457 [ENOBUFS] Insufficient resources were available in the system to complete the function.

19458 **EXAMPLES**

19459 None.

19460 **APPLICATION USAGE**

19461 None.

19462 **RATIONALE**

19463 None.

19464 **FUTURE DIRECTIONS**

19465 None.

19466 **SEE ALSO**

19467 *accept*( ), *bind*( ), *getpeername*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
19468 **<sys/socket.h>**

19469 **CHANGE HISTORY**

19470 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

19471 The **restrict** keyword is added to the *getsockname*( ) prototype for alignment with the
19472 ISO/IEC 9899: 1999 standard.

19473 **NAME**

19474 getsockopt — get the socket options

19475 **SYNOPSIS**

19476
```
#include <sys/socket.h>
```

19477
```
int getsockopt(int socket, int level, int option_name,
    void *restrict option_value, socklen_t *restrict option_len);
```

19479 **DESCRIPTION**

19480 The *getsockopt*( ) function manipulates options associated with a socket.

19481 The *getsockopt*( ) function shall retrieve the value for the option specified by the *option_name*
19482 argument for the socket specified by the *socket* argument. If the size of the option value is greater
19483 than *option_len*, the value stored in the object pointed to by the *option_value* argument shall be
19484 silently truncated. Otherwise, the object pointed to by the *option_len* argument shall be modified
19485 to indicate the actual length of the value.

19486 The *level* argument specifies the protocol level at which the option resides. To retrieve options at
19487 the socket level, specify the *level* argument as SOL_SOCKET. To retrieve options at other levels,
19488 supply the appropriate level identifier for the protocol controlling the option. For example, to
19489 indicate that an option is interpreted by the TCP (Transmission Control Protocol), set *level* to
19490 IPPROTO_TCP as defined in the **<netinet/in.h>** header.

19491 The socket in use may require the process to have appropriate privileges to use the *getsockopt*( )
19492 function.

19493 The *option_name* argument specifies a single option to be retrieved. It can be one of the following
19494 values defined in **<sys/socket.h>**:

19495 SO_DEBUG — Reports whether debugging information is being recorded. This option
19496 shall store an **int** value. This is a Boolean option.

19497 SO_ACCEPTCONN — Reports whether socket listening is enabled. This option shall store an **int**
19498 value. This is a Boolean option.

19499 SO_BROADCAST — Reports whether transmission of broadcast messages is supported, if this
19500 is supported by the protocol. This option shall store an **int** value. This is a
19501 Boolean option.

19502 SO_REUSEADDR — Reports whether the rules used in validating addresses supplied to *bind*( )
19503 should allow reuse of local addresses, if this is supported by the protocol.
19504 This option shall store an **int** value. This is a Boolean option.

19505 SO_KEEPALIVE — Reports whether connections are kept active with periodic transmission
19506 of messages, if this is supported by the protocol.

19507 If the connected socket fails to respond to these messages, the connection
19508 shall be broken and threads writing to that socket shall be notified with a
19509 SIGPIPE signal. This option shall store an **int** value. This is a Boolean
19510 option.

19511 SO_LINGER — Reports whether the socket lingers on *close*( ) if data is present. If
19512 SO_LINGER is set, the system shall block the calling thread during *close*( )
19513 until it can transmit the data or until the end of the interval indicated by
19514 the *l_linger* member, whichever comes first. If SO_LINGER is not
19515 specified, and *close*( ) is issued, the system handles the call in a way that
19516 allows the calling thread to continue as quickly as possible. This option
19517 shall store a **linger** structure.

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 61).

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.

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 1003.1-200x, **<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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/47 is applied, updating the description of SO_LINGER in the DESCRIPTION so that it blocks the calling thread rather than the process.

19596 **NAME**

19597 getsubopt — parse suboption arguments from a string

19598 **SYNOPSIS**

19599 ```
#include <stdlib.h>
```

19600 ```
int getsubopt(char **optionp, char * const *keylistp, char **valuep);
```

19601 **DESCRIPTION**

19602 The *getsubopt*( ) function shall parse suboption arguments in a flag argument. Such options often
19603 result from the use of *getopt*( ).

19604 The *getsubopt*( ) argument *optionp* is a pointer to a pointer to the option argument string. The
19605 suboption arguments shall be separated by commas and each may consist of either a single
19606 token, or a token-value pair separated by an equal sign.

19607 The *keylistp* argument shall be a pointer to a vector of strings. The end of the vector is identified
19608 by a null pointer. Each entry in the vector is one of the possible tokens that might be found in
19609 **optionp*. Since commas delimit suboption arguments in *optionp*, they should not appear in any of
19610 the strings pointed to by *keylistp*. Similarly, because an equal sign separates a token from its
19611 value, the application should not include an equal sign in any of the strings pointed to by
19612 *keylistp*.

19613 The *valuep* argument is the address of a value string pointer.

19614 If a comma appears in *optionp*, it shall be interpreted as a suboption separator. After commas
19615 have been processed, if there are one or more equal signs in a suboption string, the first equal
19616 sign in any suboption string shall be interpreted as a separator between a token and a value.
19617 Subsequent equal signs in a suboption string shall be interpreted as part of the value.

19618 If the string at **optionp* contains only one suboption argument (equivalently, no commas),
19619 *getsubopt*( ) shall update **optionp* to point to the null character at the end of the string. Otherwise,
19620 it shall isolate the suboption argument by replacing the comma separator with a null character,
19621 and shall update **optionp* to point to the start of the next suboption argument. If the suboption
19622 argument has an associated value (equivalently, contains an equal sign), *getsubopt*( ) shall update
19623 **valuep* to point to the value's first character. Otherwise, it shall set **valuep* to a null pointer. The
19624 calling application may use this information to determine whether the presence or absence of a
19625 value for the suboption is an error.

19626 Additionally, when *getsubopt*( ) fails to match the suboption argument with a token in the *keylistp*
19627 array, the calling application should decide if this is an error, or if the unrecognized option
19628 should be processed in another way.

19629 **RETURN VALUE**

19630 The *getsubopt*( ) function shall return the index of the matched token string, or –1 if no token
19631 strings were matched.

19632 **ERRORS**

19633 No errors are defined.

EXAMPLES

```
#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
};

int
main(int argc, char *argv[])
{
    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:
                        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;
...
    }
}
...
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

19730 **SEE ALSO**

19731 *getopt( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

19732 **CHANGE HISTORY**

19733 First released in Issue 4, Version 2.

19734 **Issue 5**

19735 Moved from X/OPEN UNIX extension to BASE.

19736 **Issue 6**

19737 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/26 is applied, correcting an editorial error
19738 in the SYNOPSIS.

19739 **Issue 7**

19740 The *getsubopt( )* function is moved from the XSI option to the Base.

19741 **NAME**

19742 gettimeofday — get the date and time

19743 **SYNOPSIS**

19744 OB XSI
```
#include <sys/time.h>

int gettimeofday(struct timeval *restrict tp, void *restrict tzp);
```

19746 **DESCRIPTION**

19747 The *gettimeofday*( ) function shall obtain the current time, expressed as seconds and microseconds
19748 since the Epoch, and store it in the **timeval** structure pointed to by *tp*. The resolution of the
19749 system clock is unspecified.

19750 If *tzp* is not a null pointer, the behavior is unspecified.

19751 **RETURN VALUE**

19752 The *gettimeofday*( ) function shall return 0 and no value shall be reserved to indicate an error.

19753 **ERRORS**

19754 No errors are defined.

19755 **EXAMPLES**

19756 None.

19757 **APPLICATION USAGE**

19758 Applications should use the *clock_gettime*( ) function instead of the obsolescent *gettimeofday*( )
19759 function.

19760 **RATIONALE**

19761 None.

19762 **FUTURE DIRECTIONS**

19763 The *gettimeofday*( ) function may be removed in a future version.

19764 **SEE ALSO**

19765 *clock_getres*( ), *ctime*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/time.h>**

19766 **CHANGE HISTORY**

19767 First released in Issue 4, Version 2.

19768 **Issue 5**

19769 Moved from X/OPEN UNIX extension to BASE.

19770 **Issue 6**

19771 The DESCRIPTION is updated to refer to "seconds since the Epoch" rather than "seconds since
19772 00:00:00 UTC (Coordinated Universal Time), January 1 1970" for consistency with other *time*
19773 functions.

19774 The **restrict** keyword is added to the *gettimeofday*( ) prototype for alignment with the
19775 ISO/IEC 9899: 1999 standard.

19776 **Issue 7**

19777 The *gettimeofday*( ) function is marked obsolescent.

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*( ), *setregid*( ), *setreuid*( ), *setuid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

19817 **NAME**

19818 getutxent, getutxid, getutxline — get user accounting database entries

19819 **SYNOPSIS**

19820 XSI
```
#include <utmpx.h>

struct utmpx *getutxent(void);
struct utmpx *getutxid(const struct utmpx *id);
struct utmpx *getutxline(const struct utmpx *line);
```

19824 **DESCRIPTION**

19825 Refer to *endutxent( )*.

19826 **NAME**

19827 getwc — get a wide character from a stream

19828 **SYNOPSIS**

19829 ```
#include <stdio.h>
```

19830 ```
#include <wchar.h>
```

19831 ```
wint_t getwc(FILE *stream);
```

19832 **DESCRIPTION**

19833 CX The functionality described on this reference page is aligned with the ISO C standard. Any
19834 conflict between the requirements described here and the ISO C standard is unintentional. This
19835 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

19836 The *getwc*( ) function shall be equivalent to *fgetwc*( ), except that if it is implemented as a macro it
19837 may evaluate *stream* more than once, so the argument should never be an expression with side
19838 effects.

19839 **RETURN VALUE**

19840 Refer to *fgetwc*( ).

19841 **ERRORS**

19842 Refer to *fgetwc*( ).

19843 **EXAMPLES**

19844 None.

19845 **APPLICATION USAGE**

19846 Since it may be implemented as a macro, *getwc*( ) may treat incorrectly a *stream* argument with
19847 side effects. In particular, *getwc*(*f++) does not necessarily work as expected. Therefore, use of
19848 this function is not recommended; *fgetwc*( ) should be used instead.

19849 **RATIONALE**

19850 None.

19851 **FUTURE DIRECTIONS**

19852 None.

19853 **SEE ALSO**

19854 *fgetwc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**, **<wchar.h>**

19855 **CHANGE HISTORY**

19856 First released as a World-wide Portability Interface in Issue 4. Derived from the MSE working
19857 draft.

19858 **Issue 5**

19859 The Optional Header (OH) marking is removed from **<stdio.h>**.

  
```

19860 **NAME**

19861 getwchar — get a wide character from a stdin stream

19862 **SYNOPSIS**

19863
```
#include <wchar.h>
```

19864
```
wint_t getwchar(void);
```

19865 **DESCRIPTION**

19866 CX The functionality described on this reference page is aligned with the ISO C standard. Any
19867 conflict between the requirements described here and the ISO C standard is unintentional. This
19868 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

19869 The *getwchar*( ) function shall be equivalent to *getwc*(*stdin*).

19870 **RETURN VALUE**

19871 Refer to *fgetwc*( ).

19872 **ERRORS**

19873 Refer to *fgetwc*( ).

19874 **EXAMPLES**

19875 None.

19876 **APPLICATION USAGE**

19877 If the **wint_t** value returned by *getwchar*( ) is stored into a variable of type **wchar_t** and then
19878 compared against the **wint_t** macro WEOF, the result may be incorrect. Only the **wint_t** type is
19879 guaranteed to be able to represent any wide character and WEOF.

19880 **RATIONALE**

19881 None.

19882 **FUTURE DIRECTIONS**

19883 None.

19884 **SEE ALSO**

19885 *fgetwc*( ), *getwc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

19886 **CHANGE HISTORY**

19887 First released as a World-wide Portability Interface in Issue 4. Derived from the MSE working
19888 draft.

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 1003.1-200x, Section 2.13, Pattern Matching Notation, with optional support for rule 3 in the Shell and Utilities volume of IEEE Std 1003.1-200x, 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:

Member Type	Member Name	Description
size_t	*gl_pathc*	Count of paths matched by *pattern*.
**char ** **	*gl_pathv*	Pointer to a list of matched pathnames.
size_t	*gl_offs*	Slots to reserve at the beginning of *gl_pathv*.

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 1003.1-200x, 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 ar

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 1003.1-200x, 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 1003.1-200x, 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 *errfunc* is not a null pointer, *glob*( ) calls (**errfunc*( )) with two arguments:

1. The *epath* argument is a pointer to the path that failed.
2. The *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 (**errfunc*( )) is called and returns non-zero, or if the GLOB_ERR flag is set in *flags*, *glob*( ) shall stop the scan and return GLOB_ABORTED after setting *gl_pathc* and *gl_pathv* in *pglob* to reflect the paths already scanned. If GLOB_ERR is not set and either *errfunc* is a null pointer or (**errfunc*( )) returns 0, the error shall be ignored.

The *glob*( ) function shall not fail because of large files.

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.

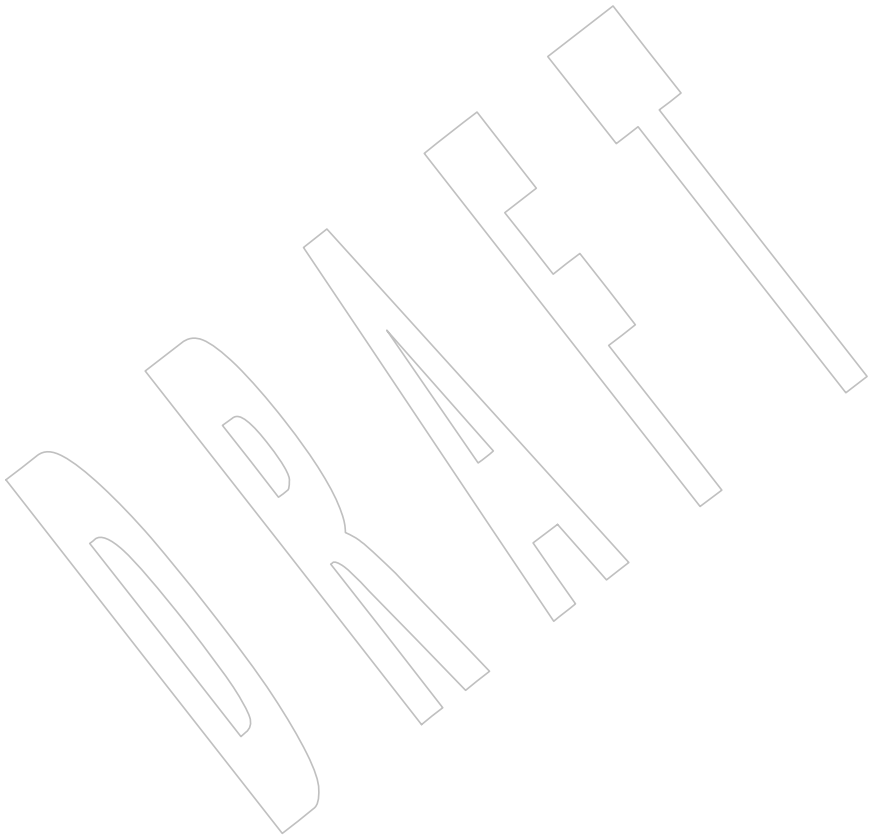
Applications that need tilde and parameter expansion should use *wordexp*( ).

RATIONALE

It was claimed that the GLOB_DOOFFS flag is unnecessary because it could be simulated using:

```
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



20052 **NAME**

20053 gmtime, gmtime_r — convert a time value to a broken-down UTC time

20054 **SYNOPSIS**

20055
```
#include <time.h>
```

20056
```
struct tm *gmtime(const time_t *timer);
```

20057 CX
```
struct tm *gmtime_r(const time_t *restrict timer,
    struct tm *restrict result);
```

20058

20059 **DESCRIPTION**

20060 CX For *gmtime*( ): The functionality described on this reference page is aligned with the ISO C
20061 standard. Any conflict between the requirements described here and the ISO C standard is
20062 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

20063 The *gmtime*( ) function shall convert the time in seconds since the Epoch pointed to by *timer* into
20064 a broken-down time, expressed as Coordinated Universal Time (UTC).

20065 CX The relationship between a time in seconds since the Epoch used as an argument to *gmtime*( )
20066 and the **tm** structure (defined in the **<time.h>** header) is that the result shall be as specified in
20067 the expression given in the definition of seconds since the Epoch (see the Base Definitions
20068 volume of IEEE Std 1003.1-200x, Section 4.14, Seconds Since the Epoch), where the names in the
20069 structure and in the expression correspond.

20070 The same relationship shall apply for *gmtime_r*( ).

20071 The *gmtime*( ) function need not be thread-safe. A function that is not required to be thread-safe
20072 is not required to be reentrant.

20073 The *asctime*( ), *ctime*( ), *gmtime*( ), and *localtime*( ) functions shall return values in one of two static
20074 objects: a broken-down time structure and an array of type **char**. Execution of any of the
20075 functions may overwrite the information returned in either of these objects by any of the other
20076 functions.

20077 The *gmtime_r*( ) function shall convert the time in seconds since the Epoch pointed to by *timer*
20078 into a broken-down time expressed as Coordinated Universal Time (UTC). The broken-down
20079 time is stored in the structure referred to by *result*. The *gmtime_r*( ) function shall also return the
20080 address of the same structure.

20081 **RETURN VALUE**

20082 Upon successful completion, the *gmtime*( ) function shall return a pointer to a **struct tm**. If an
20083 CX error is detected, *gmtime*( ) shall return a null pointer and set *errno* to indicate the error.

20084 Upon successful completion, *gmtime_r*( ) shall return the address of the structure pointed to by
20085 the argument *result*. If an error is detected, *gmtime_r*( ) shall return a null pointer and set *errno* to
20086 indicate the error.

20087 **ERRORS**

20088 CX The *gmtime*( ) and *gmtime_r*( ) functions shall fail if:

20089 CX [EOVERFLOW] The result cannot be represented.

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 1003.1-200x, **<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.

The *gmtime_r*( ) function is included for alignment with the POSIX Threads Extension.

Issue 6

Th*gmtime*e *released in Issue 1. D.l(DESCe8gLment8is included for tcth t D.oiESCe8gLment8is included c56 6c /R12 100 -(i3 B)*

20121 **NAME**

20122 grantpt — grant access to the slave pseudo-terminal device

20123 **SYNOPSIS**

20124 XSI
```
#include <stdlib.h>

int grantpt(int fildes);
```

20126 **DESCRIPTION**

20127 The *grantpt*( ) function shall change the mode and ownership of the slave pseudo-terminal
20128 device associated with its master pseudo-terminal counterpart. The *fildes* argument is a file
20129 descriptor that refers to a master pseudo-terminal device. The user ID of the slave shall be set to
20130 the real UID of the calling process and the group ID shall be set to an unspecified group ID. The
20131 permission mode of the slave pseudo-terminal shall be set to readable and writable by the
20132 owner, and writable by the group.

20133 The behavior of the *grantpt*( ) function is unspecified if the application has installed a signal
20134 handler to catch SIGCHLD signals.

20135 **RETURN VALUE**

20136 Upon successful completion, *grantpt*( ) shall return 0; otherwise, it shall return –1 and set *errno* to
20137 indicate the error.

20138 **ERRORS**

20139 The *grantpt*( ) function may fail if:

20140 [EBADF] The *fildes* argument is not a valid open file descriptor.

20141 [EINVAL] The *fildes* argument is not associated with a master pseudo-terminal device.

20142 [EACCES] The corresponding slave pseudo-terminal device could not be accessed.

20143 **EXAMPLES**

20144 None.

20145 **APPLICATION USAGE**

20146 None.

20147 **RATIONALE**

20148 None.

20149 **FUTURE DIRECTIONS**

20150 None.

20151 **SEE ALSO**

20152 *open*( ), *ptsname*( ), *unlockpt*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

20153 **CHANGE HISTORY**

20154 First released in Issue 4, Version 2.

20155 **Issue 5**

20156 Moved from X/OPEN UNIX extension to BASE.

20157 The last paragraph of the DESCRIPTION is moved from the APPLICATION USAGE section.

20158 **NAME**

20159 hcreate, hdestroy, hsearch — manage hash search table

20160 **SYNOPSIS**

20161 XSI
```
#include <search.h>

int hcreate(size_t nel);
void hdestroy(void);
ENTRY *hsearch(ENTRY item, ACTION action);
```

20165 **DESCRIPTION**

20166 The *hcreate*( ), *hdestroy*( ), and *hsearch*( ) functions shall manage hash search tables.

20167 The *hcreate*( ) function shall allocate sufficient space for the table, and the application shall
20168 ensure it is called before *hsearch*( ) is used. The *nel* argument is an estimate of the maximum
20169 number of entries that the table shall contain. This number may be adjusted upward by the
20170 algorithm in order to obtain certain mathematically favorable circumstances.

20171 The *hdestroy*( ) function shall dispose of the search table, and may be followed by another call to
20172 *hcreate*( ). After the call to *hdestroy*( ), the data can no longer be considered accessible.

20173 The *hsearch*( ) function is a hash-table search routine. It shall return a pointer into a hash table
20174 indicating the location at which an entry can be found. The *item* argument is a structure of type
20175 **ENTRY** (defined in the **<search.h>** header) containing two pointers: *item.key* points to the
20176 comparison key (a **char** *), and *item.data* (a **void** *) points to any other data to be associated with
20177 that key. The comparison function used by *hsearch*( ) is *strcmp*( ). The *action* argument is a
20178 member of an enumeration type **ACTION** indicating the disposition of the entry if it cannot be
20179 found in the table. ENTER indicates that the item should be inserted in the table at an
20180 appropriate point. FIND indicates that no entry should be made. Unsuccessful resolution is
20181 indicated by the return of a null pointer.

20182 These functions need not be thread-safe. A function that is not required to be thread-safe is not
20183 required to be reentrant.

20184 **RETURN VALUE**

20185 The *hcreate*( ) function shall return 0 if it cannot allocate sufficient space for the table; otherwise,
20186 it shall return non-zero.

20187 The *hdestroy*( ) function shall not return a value.

20188 The *hsearch*( ) function shall return a null pointer if either the action is FIND and the item could
20189 not be found or the action is ENTER and the table is full.

20190 **ERRORS**

20191 The *hcreate*( ) and *hsearch*( ) functions may fail if:

20192 [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.

```
#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. */
};

#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;
}
```


20241 **APPLICATION USAGE**

20242 The *hcreate*( ) and *hsearch*( ) functions may use *malloc*( ) to allocate space.

20243 **RATIONALE**

20244 None.

20245 **FUTURE DIRECTIONS**

20246 None.

20247 **SEE ALSO**

20248 *bsearch*( ), *lsearch*( ), *malloc*( ), *strcmp*( ), *tsearch*( ), the Base Definitions volume of
20249 IEEE Std 1003.1-200x, **<search.h>**

20250 **CHANGE HISTORY**

20251 First released in Issue 1. Derived from Issue 1 of the SVID.

20252 **Issue 6**

20253 The normative text is updated to avoid use of the term "must" for application requirements.

20254 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

20255 **NAME**

20256 htonl, htons, ntohl, ntohs — convert values between host and network byte order

20257 **SYNOPSIS**

20258
```
#include <arpa/inet.h>
```

20259
```
uint32_t htonl(uint32_t hostlong);
```
20260
```
uint16_t htons(uint16_t hostshort);
```
20261
```
uint32_t ntohl(uint32_t netlong);
```
20262
```
uint16_t ntohs(uint16_t netshort);
```

20263 **DESCRIPTION**

20264 These functions shall convert 16-bit and 32-bit quantities between network byte order and host
20265 byte order.

20266 On some implementations, these functions are defined as macros.

20267 The **uint32_t** and **uint16_t** types are defined in **<inttypes.h>**.

20268 **RETURN VALUE**

20269 The *htonl*( ) and *htons*( ) functions shall return the argument value converted from host to
20270 network byte order.

20271 The *ntohl*( ) and *ntohs*( ) functions shall return the argument value converted from network to
20272 host byte order.

20273 **ERRORS**

20274 No errors are defined.

20275 **EXAMPLES**

20276 None.

20277 **APPLICATION USAGE**

20278 These functions are most often used in conjunction with IPv4 addresses and ports as returned by
20279 *gethostent*( ) and *getservent*( ).

20280 **RATIONALE**

20281 None.

20282 **FUTURE DIRECTIONS**

20283 None.

20284 **SEE ALSO**

20285 *endhostent*( ), *endservent*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<inttypes.h>**,
20286 **<arpa/inet.h>**

20287 **CHANGE HISTORY**

20288 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

20289 **NAME**

20290 hypot, hypotf, hypotl — Euclidean distance function

20291 **SYNOPSIS**

20292
```
#include <math.h>
```

20293
```
double hypot(double x, double y);
```
20294
```
float hypotf(float x, float y);
```
20295
```
long double hypotl(long double x, long double y);
```

20296 **DESCRIPTION**

20297 CX The functionality described on this reference page is aligned with the ISO C standard. Any
20298 conflict between the requirements described here and the ISO C standard is unintentional. This
20299 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

20300 These functions shall compute the value of the square root of x^2+y^2 without undue overflow or
20301 underflow.

20302 An application wishing to check for error situations should set *errno* to zero and call
20303 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
20304 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
20305 zero, an error has occurred.

20306 **RETURN VALUE**

20307 Upon successful completion, these functions shall return the length of the hypotenuse of a right-
20308 angled triangle with sides of length *x* and *y*.

20309 If the correct value would cause overflow, a range error shall occur and *hypot*( ), *hypotf*( ), and
20310 *hypotl*( ) shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL,
20311 respectively.

20312 MX If *x* or *y* is ±Inf, +Inf shall be returned (even if one of *x* or *y* is NaN).

20313 If *x* or *y* is NaN, and the other is not ±Inf, a NaN shall be returned.

20314 If both arguments are subnormal and the correct result is subnormal, a range error may occur
20315 and the correct result is returned.

20316 **ERRORS**

20317 These functions shall fail if:

20318 Range Error The result overflows.

20319 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
20320 then *errno* shall be set to [ERANGE]. If the integer expression
20321 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
20322 floating-point exception shall be raised.

20323 These functions may fail if:

20324 MX Range Error The result underflows.

20325 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
20326 then *errno* shall be set to [ERANGE]. If the integer expression
20327 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
20328 floating-point exception shall be raised.

EXAMPLES

See the EXAMPLES section in *atan2( )*.

APPLICATION USAGE

hypot(*x*,*y*), *hypot*(*y*,*x*), and *hypot*(*x*, –*y*) are equivalent.

hypot(*x*, ±0) is equivalent to *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 (*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( )*, *sqrt( )*, the Base Definitions volume of IEEE Std 1003.1-200x, 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 *hypot( )* function is no longer marked as an extension.

The *hypotf( )* and *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.

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

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/49 is applied, updating the EXAMPLES section.

20362 **NAME**

20363 iconv — codeset conversion function

20364 **SYNOPSIS**

20365
```
#include <iconv.h>
```

20366
```
size_t iconv(iconv_t cd, char **restrict inbuf,
    size_t *restrict inbytesleft, char **restrict outbuf,
    size_t *restrict outbytesleft);
```

20369 **DESCRIPTION**

20370 The *iconv*( ) function shall convert the sequence of characters from one codeset, in the array
20371 specified by *inbuf*, into a sequence of corresponding characters in another codeset, in the array
20372 specified by *outbuf*. The codesets are those specified in the *iconv_open*( ) call that returned the
20373 conversion descriptor, *cd*. The *inbuf* argument points to a variable that points to the first
20374 character in the input buffer and *inbytesleft* indicates the number of bytes to the end of the buffer
20375 to be converted. The *outbuf* argument points to a variable that points to the first available byte in
20376 the output buffer and *outbytesleft* indicates the number of the available bytes to the end of the
20377 buffer.

20378 For state-dependent encodings, the conversion descriptor *cd* is placed into its initial shift state by
20379 a call for which *inbuf* is a null pointer, or for which *inbuf* points to a null pointer. When *iconv*( ) is
20380 called in this way, and if *outbuf* is not a null pointer or a pointer to a null pointer, and *outbytesleft*
20381 points to a positive value, *iconv*( ) shall place, into the output buffer, the byte sequence to change
20382 the output buffer to its initial shift state. If the output buffer is not large enough to hold the
20383 entire reset sequence, *iconv*( ) shall fail and set *errno* to [E2BIG]. Subsequent calls with *inbuf* as
20384 other than a null pointer or a pointer to a null pointer cause the conversion to take place from
20385 the current state of the conversion descriptor.

20386 If a sequence of input bytes does not form a valid character in the specified codeset, conversion
20387 shall stop after the previous successfully converted character. If the input buffer ends with an
20388 incomplete character or shift sequence, conversion shall stop after the previous successfully
20389 converted bytes. If the output buffer is not large enough to hold the entire converted input,
20390 conversion shall stop just prior to the input bytes that would cause the output buffer to
20391 overflow. The variable pointed to by *inbuf* shall be updated to point to the byte following the last
20392 byte successfully used in the conversion. The value pointed to by *inbytesleft* shall be
20393 decremented to reflect the number of bytes still not converted in the input buffer. The variable
20394 pointed to by *outbuf* shall be updated to point to the byte following the last byte of converted
20395 output data. The value pointed to by *outbytesleft* shall be decremented to reflect the number of
20396 bytes still available in the output buffer. For state-dependent encodings, the conversion
20397 descriptor shall be updated to reflect the shift state in effect at the end of the last successfully
20398 converted byte sequence.

20399 If *iconv*( ) encounters a character in the input buffer that is valid, but for which an identical
20400 character does not exist in the target codeset, *iconv*( ) shall perform an implementation-defined
20401 conversion on this character.

20402 **RETURN VALUE**

20403 The *iconv*( ) function shall update the variables pointed to by the arguments to reflect the extent
20404 of the conversion and return the number of non-identical conversions performed. If the entire
20405 string in the input buffer is converted, the value pointed to by *inbytesleft* shall be 0. If the input
20406 conversion is stopped due to any conditions mentioned above, the value pointed to by *inbytesleft*
20407 shall be non-zero and *errno* shall be set to indicate the condition. If an error occurs, *iconv*( ) shall
20408 return (**size_t**)–1 and set *errno* to indicate the error.

ERRORS

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 *intbytesleft* 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*( ), *mbsrtowcs*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<iconv.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the HP-UX Manual.

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.

Issue 7

The *iconv*( ) function is moved from the XSI option to the Base.

20456 **NAME**

20457 iconv_close — codeset conversion deallocation function

20458 **SYNOPSIS**

20459
```
#include <iconv.h>
```

20460
```
int iconv_close(iconv_t cd);
```

20461 **DESCRIPTION**

20462 The *iconv_close*( ) function shall deallocate the conversion descriptor *cd* and all other associated
20463 resources allocated by *iconv_open*( ).

20464 If a file descriptor is used to implement the type **iconv_t**, that file descriptor shall be closed.

20465 **RETURN VALUE**

20466 Upon successful completion, 0 shall be returned; otherwise, –1 shall be returned and *errno* set to
20467 indicate the error.

20468 **ERRORS**

20469 The *iconv_close*( ) function may fail if:

20470 [EBADF] The conversion descriptor is invalid.

20471 **EXAMPLES**

20472 None.

20473 **APPLICATION USAGE**

20474 None.

20475 **RATIONALE**

20476 None.

20477 **FUTURE DIRECTIONS**

20478 None.

20479 **SEE ALSO**

20480 *iconv*( ), *iconv_open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<iconv.h>**

20481 **CHANGE HISTORY**

20482 First released in Issue 4. Derived from the HP-UX Manual.

20483 **Issue 7**

20484 The *iconv_close*( ) function is moved from the XSI option to the Base.

NAME

iconv_open — codeset conversion allocation function

SYNOPSIS

```
#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] All file descriptors available to the process are currently open.

[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.

20525 **SEE ALSO**

20526 *iconv( )*, *iconv_close( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<fcntl.h>**, **<iconv.h>**

20527 **CHANGE HISTORY**

20528 First released in Issue 4. Derived from the HP-UX Manual.

20529 **Issue 7**

20530 SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

20531 The *iconv_open( )* function is moved from the XSI option to the Base.

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 1003.1-200x, **<net/if.h>**

CHANGE HISTORY

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

20558 **NAME**

20559 if_indextoname — map a network interface index to its corresponding name

20560 **SYNOPSIS**

20561 ```
#include <net/if.h>
```

20562 ```
char *if_indextoname(unsigned ifindex, char *ifname);
```

20563 **DESCRIPTION**

20564 The *if_indextoname*( ) function shall map an interface index to its corresponding name.

20565 When this function is called, *ifname* shall point to a buffer of at least {IF_NAMESIZE} bytes. The
20566 function shall place in this buffer the name of the interface with index *ifindex*.

20567 **RETURN VALUE**

20568 If *ifindex* is an interface index, then the function shall return the value supplied in *ifname*, which
20569 points to a buffer now containing the interface name. Otherwise, the function shall return a
20570 NULL pointer and set *errno* to indicate the error.

20571 **ERRORS**

20572 The *if_indextoname*( ) function shall fail if:

20573 [ENXIO] The interface does not exist.

20574 **EXAMPLES**

20575 None.

20576 **APPLICATION USAGE**

20577 None.

20578 **RATIONALE**

20579 None.

20580 **FUTURE DIRECTIONS**

20581 None.

20582 **SEE ALSO**

20583 *getsockopt*( ), *if_freenameindex*( ), *if_nameindex*( ), *if_nametoindex*( ), *setsockopt*( ), the Base Definitions
20584 volume of IEEE Std 1003.1-200x, **<net/if.h>**

20585 **CHANGE HISTORY**

20586 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

20587 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/28 is applied, changing {IFNAMSIZ} to
20588 {IF_NAMESIZ} in the DESCRIPTION.

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_freenameindex*( ) 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

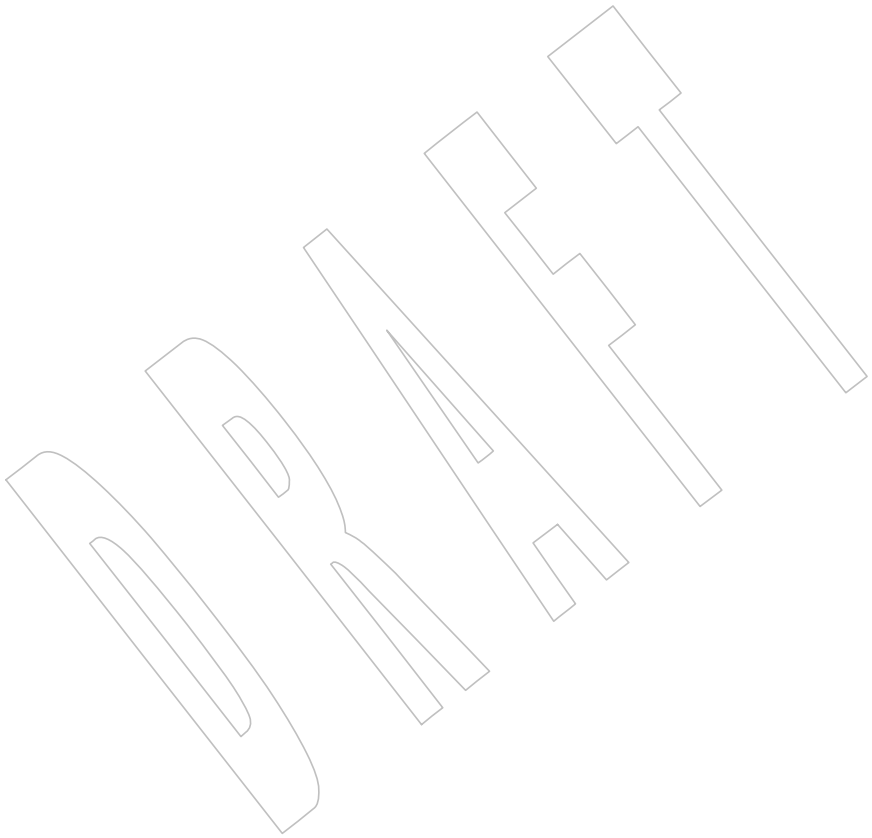
None.

SEE ALSO

getsockopt( ), *if_freenameindex*( ), *if_indextoname*( ), *if_nametoindex*( ), *setsockopt*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<net/if.h>**

CHANGE HISTORY

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



ilogb( )

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

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 1003.1-200x 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 *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 exponent part of *x* as a signed integer value. They are equivalent to calling the corresponding *logb*( ) function and casting the returned value to type **int**.

XSI If *x* is 0, the value FP_ILOGB0 shall be returned. On XSI-conformant systems, a domain error shall occur;

CX otherwise, a domain error may occur.

XSI If *x* is ±Inf, the value {INT_MAX} shall be returned. On XSI-conformant systems, a domain error shall occur;

CX otherwise, a domain error may occur.

XSI If *x* is a NaN, the value FP_ILOGBNAN shall be returned.

shall be raised.

These functions may fail if:

Domain Error — The *x* argument is zero, NaN, or ±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.

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 LDBL_MAX_EXP*log2(FLT_RADIX) or {INT_MIN} must be greater than LDBL_MIN_EXP*log2(FLT_RADIX) if subnormals are not supported, or {INT_MIN} must be greater than (LDBL_MIN_EXP-LDBL_MANT_DIG)*log2(FLT_RADIX) if subnormals are supported.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept( )*, *logb( )*, *scalbln( )*, the Base Definitions volume of IEEE Std 1003.1-200x, 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.

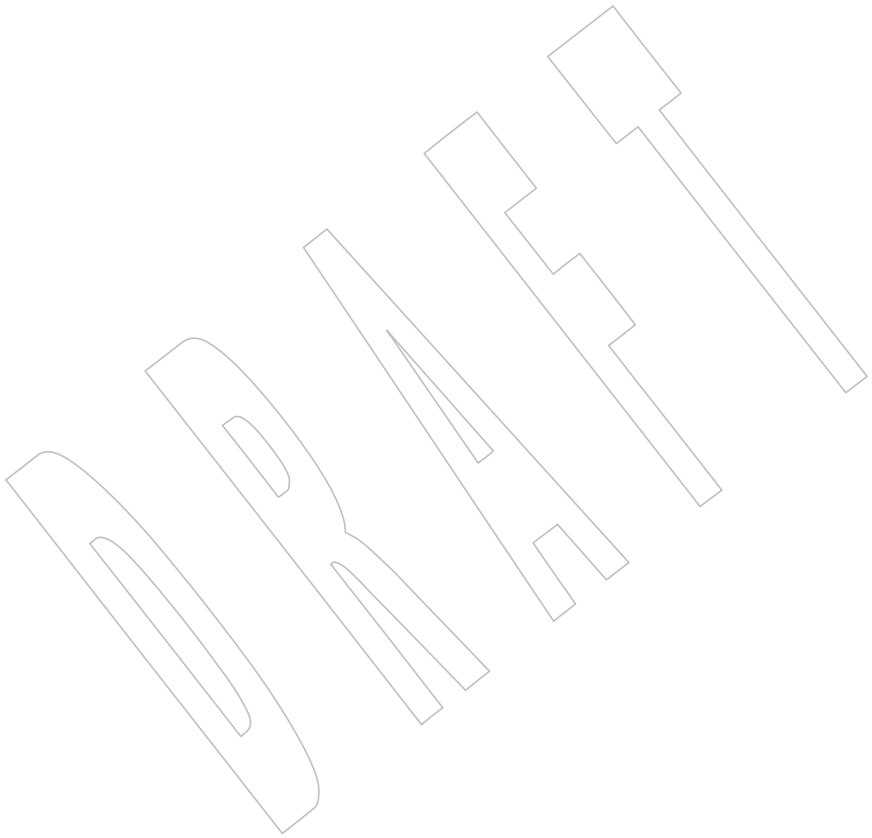
The RETURN VALUE section is revised for alignment with the ISO/IEC 9899: 1999 standard.

Functionality relating to the XSI option is marked.

Issue 7

ISO/IEC 9899: 1999 standard, Technical Corrigendum 2 #48 (SD5-XSH-ERN-71), #49, and #79 (SD5-XSH-ERN-72) are applied.

imaxabs( ) *System Interfaces*



20754 **NAME**

20755 imaxdiv — return quotient and remainder

20756 **SYNOPSIS**

20757
```
#include <inttypes.h>
```

20758
```
imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
```

20759 **DESCRIPTION**

20760 CX The functionality described on this reference page is aligned with the ISO C standard. Any
20761 conflict between the requirements described here and the ISO C standard is unintentional. This
20762 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

20763 The *imaxdiv*( ) function shall compute *numer* / *denom* and *numer* % *denom* in a single operation.

20764 **RETURN VALUE**

20765 The *imaxdiv*( ) function shall return a structure of type **imaxdiv_t**, comprising both the quotient
20766 and the remainder. The structure shall contain (in either order) the members *quot* (the quotient)
20767 and *rem* (the remainder), each of which has type **intmax_t**.

20768 If either part of the result cannot be represented, the behavior is undefined.

20769 **ERRORS**

20770 No errors are defined.

20771 **EXAMPLES**

20772 None.

20773 **APPLICATION USAGE**

20774 None.

20775 **RATIONALE**

20776 None.

20777 **FUTURE DIRECTIONS**

20778 None.

20779 **SEE ALSO**

20780 *imaxabs*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<inttypes.h>**

20781 **CHANGE HISTORY**

20782 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

inet_addr( )

NAME

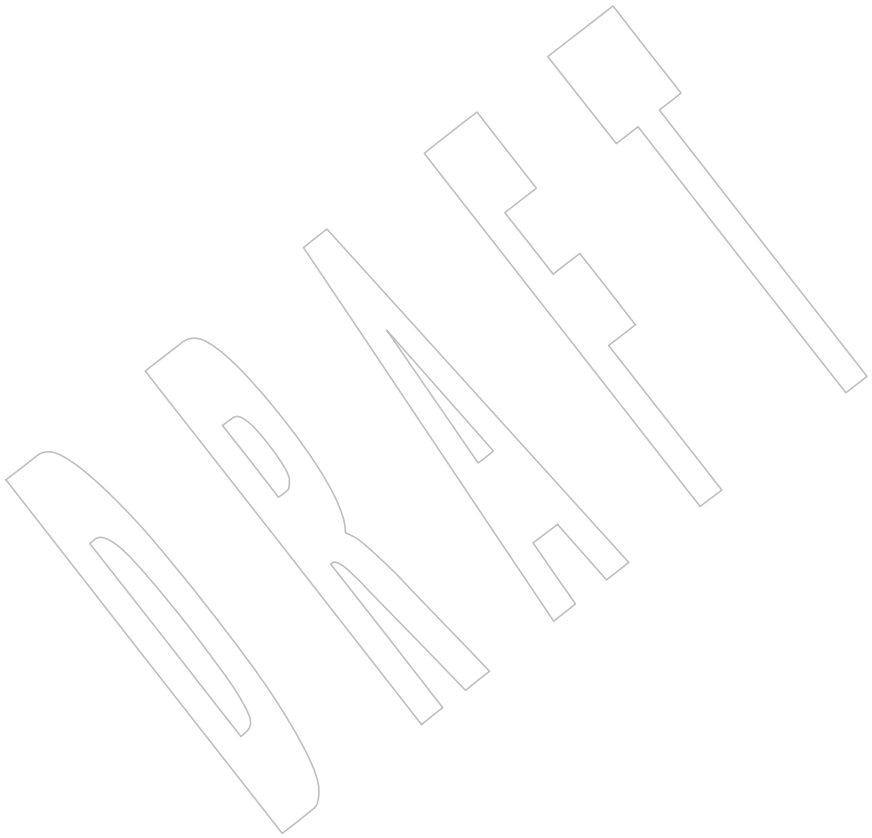
inet_addr, inet_ntoa — IPv4 address manipulation

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 t



20820 **EXAMPLES**

20821 None.

20822 **APPLICATION USAGE**

20823 The return value of *inet_ntoa*( ) may point to static data that may be overwritten by subsequent
20824 calls to *inet_ntoa*( ).

20825 **RATIONALE**

20826 None.

20827 **FUTURE DIRECTIONS**

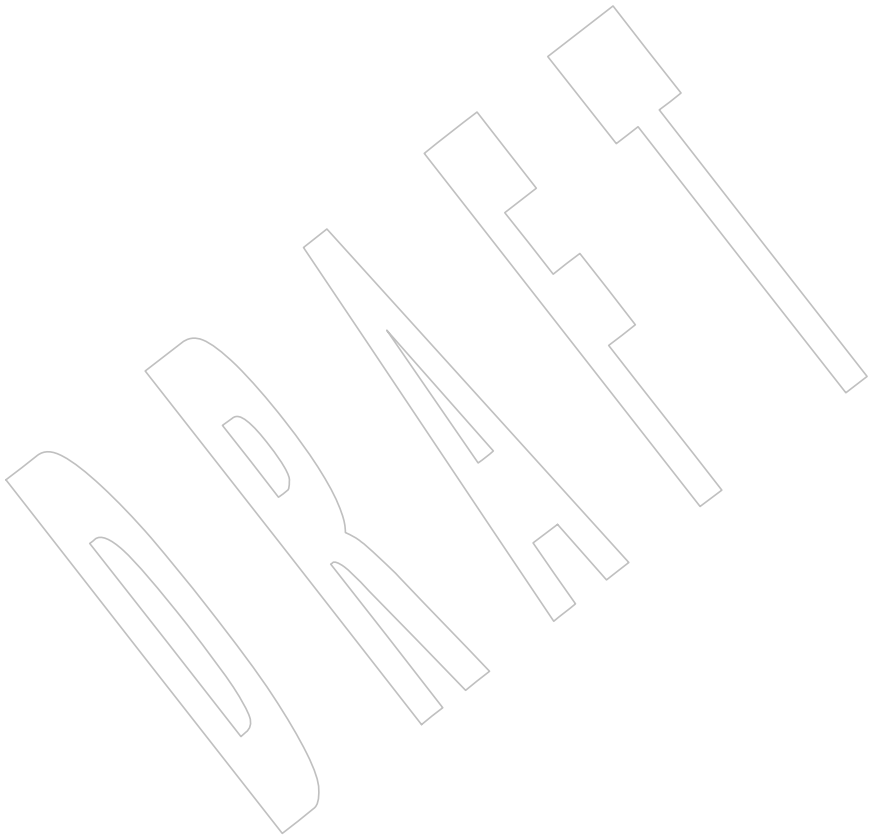
20828 None.

20829 **SEE ALSO**

20830 *endhostent*( ), *endnetent*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<arpa/inet.h>**

20831 **CHANGE HISTORY**

20832 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.



20833 **NAME**

20834 inet_ntop, inet_pton — convert IPv4 and IPv6 addresses between binary and text form

20835 **SYNOPSIS**

20836
```
#include <arpa/inet.h>
```

20837
```
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);
```

20840 **DESCRIPTION**

20841 The *inet_ntop*( ) function shall convert a numeric address into a text string suitable for
20842 IP6 presentation. The *af* argument shall specify the family of the address. This can be AF_INET or
20843 AF_INET6. The *src* argument points to a buffer holding an IPv4 address if the *af* argument is
20844 IP6 AF_INET, or an IPv6 address if the *af* argument is AF_INET6; the address must be in network
20845 byte order. The *dst* argument points to a buffer where the function stores the resulting text string;
20846 it shall not be NULL. The *size* argument specifies the size of this buffer, which shall be large
20847 IP6 enough to hold the text string (INET_ADDRSTRLEN characters for IPv4,
20848 INET6_ADDRSTRLEN characters for IPv6).

20849 The *inet_pton*( ) function shall convert an address in its standard text presentation form into its
20850 IP6 numeric binary form. The *af* argument shall specify the family of the address. The AF_INET and
20851 AF_INET6 address families shall be supported. The *src* argument points to the string being
20852 passed in. The *dst* argument points to a buffer into which the function stores the numeric
20853 IP6 address; this shall be large enough to hold the numeric address (32 bits for AF_INET, 128 bits
20854 for AF_INET6).

20855 If the *af* argument of *inet_pton*( ) is AF_INET, the *src* string shall be in the standard IPv4 dotted-
20856 decimal form:

20857
```
ddd.ddd.ddd.ddd
```

20858 where `"ddd"` is a one to three digit decimal number between 0 and 255 (see *inet_addr*( )). The
20859 *inet_pton*( ) function does not accept other formats (such as the octal numbers, hexadecimal
20860 numbers, and fewer than four numbers that *inet_addr*( ) accepts).

20861 IP6 If the *af* argument of *inet_pton*( ) is AF_INET6, the *src* string shall be in one of the following
20862 standard IPv6 text forms:

20863 1. The preferred form is `"x:x:x:x:x:x:x:x"`, where the `'x'`s are the hexadecimal values
20864 of the eight 16-bit pieces of the address. Leading zeros in individual fields can be
20865 omitted, but there shall be at least one numeral in every field.

20866 2. A string of contiguous zero fields in the preferred form can be shown as `"::"`. The `"::"`
20867 can only appear once in an address. Unspecified addresses (`"0:0:0:0:0:0:0:0"`) may
20868 be represented simply as `"::"`.

20869 3. A third form that is sometimes more convenient when dealing with a mixed environment
20870 of IPv4 and IPv6 nodes is `"x:x:x:x:x:x:d.d.d.d"`, where the `'x'`s are the
20871 hexadecimal values of the six high-order 16-bit pieces of the address, and the `'d'`s are the
20872 decimal values of the four low-order 8-bit pieces of the address (standard IPv4
20873 representation).

20874 **Note:** A more extensive description of the standard representations of IPv6 addresses can be found in
20875 RFC 2373.

20876 **RETURN VALUE**

20877 The *inet_ntop*( ) function shall return a pointer to the buffer containing the text string if the
20878 conversion succeeds, and NULL otherwise, and set *errno* to indicate the error.

20879 The *inet_pton*( ) function shall return 1 if the conversion succeeds, with the address pointed to by
20880 IP6 *dst* in network byte order. It shall return 0 if the input is not a valid IPv4 dotted-decimal string
20881 or a valid IPv6 address string, or −1 with *errno* set to [EAFNOSUPPORT] if the *af* argument is
20882 unknown.

20883 **ERRORS**

20884 The *inet_ntop*( ) and *inet_pton*( ) functions shall fail if:

20885 [EAFNOSUPPORT]
20886 The *af* argument is invalid.

20887 [ENOSPC] The size of the *inet_ntop*( ) result buffer is inadequate.

20888 **EXAMPLES**

20889 None.

20890 **APPLICATION USAGE**

20891 None.

20892 **RATIONALE**

20893 None.

20894 **FUTURE DIRECTIONS**

20895 None.

20896 **SEE ALSO**

20897 The Base Definitions volume of IEEE Std 1003.1-200x, **<arpa/inet.h>**

20898 **CHANGE HISTORY**

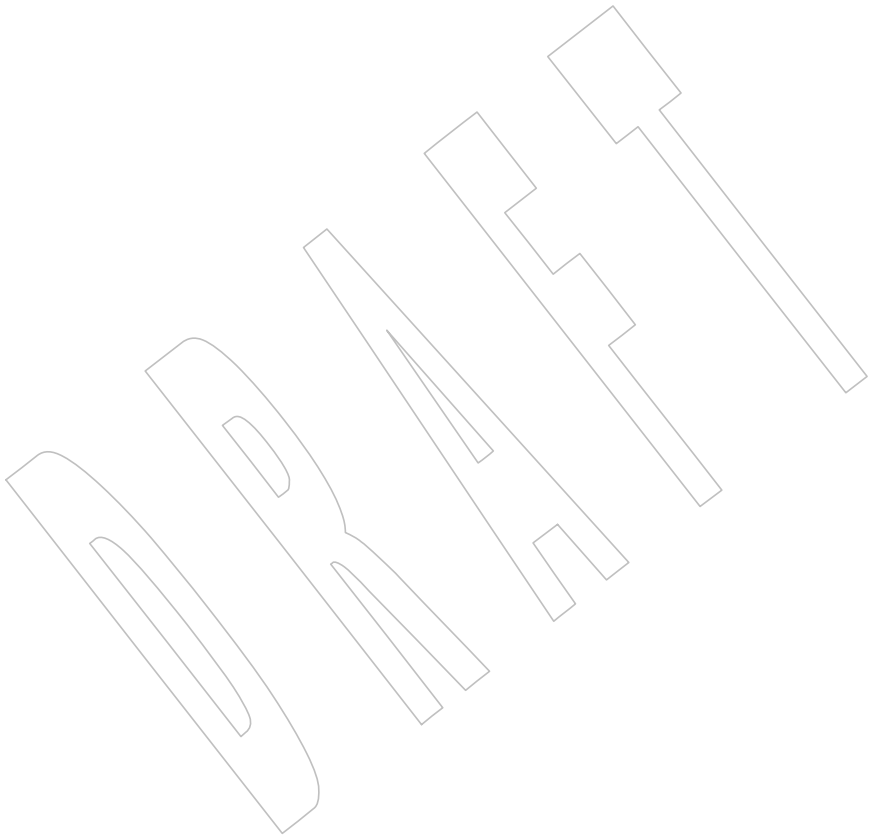
20899 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

20900 IPv6 extensions are marked.

20901 The **restrict** keyword is added to the *inet_ntop*( ) and *inet_pton*( ) prototypes for alignment with
20902 the ISO/IEC 9899: 1999 standard.

20903 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/29 is applied, adding "the address must
20904 be in network byte order" to the end of the fourth sentence of the first paragraph in the
20905 DESCRIPTION.

initstate( )



20947 **ERRORS**
20948 No errors are defined.

20949 **EXAMPLES**
20950 None.

20951 **APPLICATION USAGE**
20952 After initialization, a state array can be restarted at a different point in one of two ways:

20953 1. The *initstate*( ) function can be used, with the desired seed, state array, and size of the
20954 array.

20955 2. The *setstate*( ) function, with the desired state, can be used, followed by *srandom*( ) with
20956 the desired seed. The advantage of using both of these functions is that the size of the
20957 state array does not have to be saved once it is initialized.

20958 Although some implementations of *random*( ) have written messages to standard error, such
20959 implementations do not conform to IEEE Std 1003.1-200x.

20960 Issue 5 restored the historical behavior of this function.

20961 Threaded applications should use *erand48*( ), *nrand48*( ), or *jrand48*( ) instead of *random*( ) when
20962 an independent random number sequence in multiple threads is required.

20963 **RATIONALE**
20964 None.

20965 **FUTURE DIRECTIONS**
20966 None.

20967 **SEE ALSO**
20968 *drand48*( ), *rand*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

20969 **CHANGE HISTORY**
20970 First released in Issue 4, Version 2.

20971 **Issue 5**
20972 Moved from X/OPEN UNIX extension to BASE.

20973 In the DESCRIPTION, the phrase "values smaller than 8" is replaced with "values greater than
20974 or equal to 8, or less than 32", "*size*<8" is replaced with "8≤*size* <32", and a new first paragraph
20975 is added to the RETURN VALUE section. A note is added to the APPLICATION USAGE
20976 indicating that these changes restore the historical behavior of the function.

20977 **Issue 6**
20978 In the DESCRIPTION, duplicate text "For values greater than or equal to 8 . . ." is removed.

20979 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/30 is applied, removing *rand_r*( ) from the
20980 list of suggested functions in the APPLICATION USAGE section.

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);
```

21023 **Creating a Circular Linked List**

21024 The following example creates a circular linked list.

```
21025 #include <search.h>
21026 ...
21027 struct myque element1;
21028 struct myque element2;

21029 char *data1 = "DATA1";
21030 char *data2 = "DATA2";
21031 ...
21032 element1.data = data1;
21033 element2.data = data2;

21034 element1.fwd = &element1;
21035 element1.bck = &element1;

21036 insque (&element2, &element1);
```

21037 **Removing an Element**

21038 The following example removes the element pointed to by *element1*.

```
21039 #include <search.h>
21040 ...
21041 struct myque element1;
21042 ...
21043 remque (&element1);
```

21044 **APPLICATION USAGE**

21045 The historical implementations of these functions described the arguments as being of type
21046 **struct qelem** * rather than as being of type **void** * as defined here. In those implementations,
21047 **struct qelem** was commonly defined in **<search.h>** as:

```
21048 struct qelem {
21049     struct qelem  *q_forw;
21050     struct qelem  *q_back;
21051 };
```

21052 Applications using these functions, however, were never able to use this structure directly since
21053 it provided no room for the actual data contained in the elements. Most applications defined
21054 structures that contained the two pointers as the initial elements and also provided space for, or
21055 pointers to, the object's data. Applications that used these functions to update more than one
21056 type of table also had the problem of specifying two or more different structures with the same
21057 name, if they literally used **struct qelem** as specified.

21058 As described here, the implementations were actually expecting a structure type where the first
21059 two members were forward and backward pointers to structures. With C compilers that didn't
21060 provide function prototypes, applications used structures as specified in the DESCRIPTION
21061 above and the compiler did what the application expected.

21062 If this method had been carried forward with an ISO C standard compiler and the historical
21063 function prototype, most applications would have to be modified to cast pointers to the
21064 structures actually used to be pointers to **struct qelem** to avoid compilation warnings. By
21065 specifying **void** * as the argument type, applications do not need to change (unless they
21066 specifically referenced **struct qelem** and depended on it being defined in **<search.h>**).

21067 **RATIONALE**

21068 None.

21069 **FUTURE DIRECTIONS**

21070 None.

21071 **SEE ALSO**

21072 The Base Definitions volume of IEEE Std 1003.1-200x, **<search.h>**

21073 **CHANGE HISTORY**

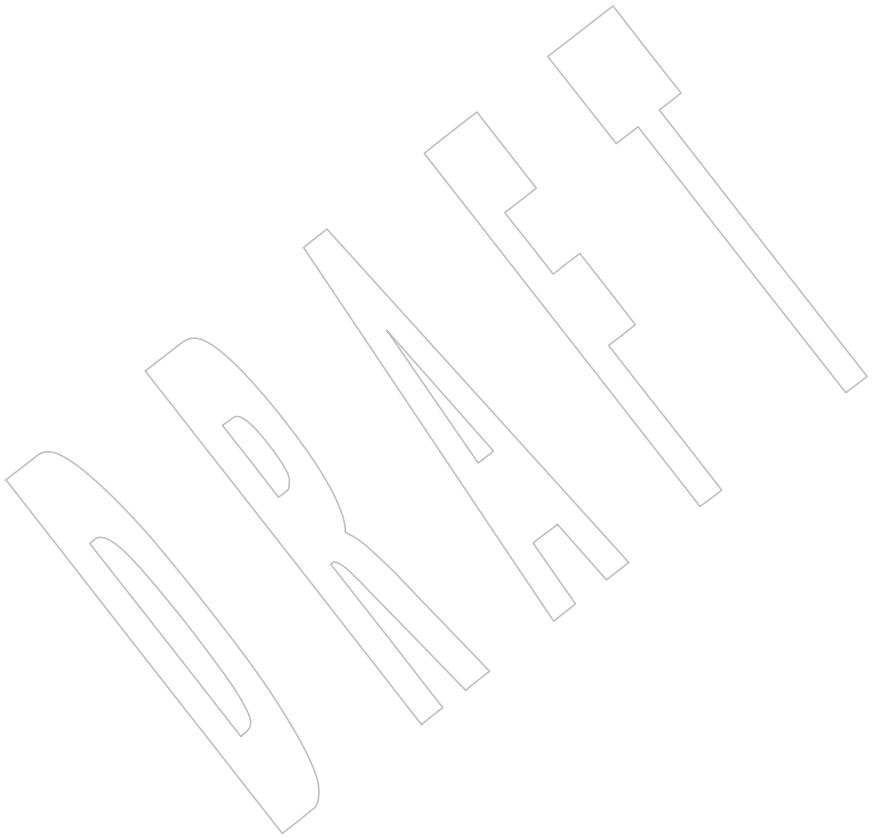
21074 First released in Issue 4, Version 2.

21075 **Issue 5**

21076 Moved from X/OPEN UNIX extension to BASE.

21077 **Issue 6**

21078 The normative text is updated to avoid use of the term "must" for application requirements.



NAME

ioctl — control a STREAMS device (**STREAMS**)

SYNOPSIS

OB 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* ar

I_FLUSH Flushes read and/or write queues, depending on the value of *arg*. Valid *arg* values are:

- FLUSHR Flush all read queues.
- FLUSHW Flush all write queues.
- FLUSHRW Flush all read and all write queues.

The *ioctl*( ) function with the I_FLUSH command shall fail if:

- [EINVAL] Invalid *arg* value.
- [EAGAIN] or [ENOSR] Unable to allocate buffers for flush message.
- [ENXIO] Hangup received on *fildes*.

I_FLUSHBAND Flushes a particular band of messages. The *arg* argument points to a **bandinfo** structure. The *bi_flag* member may be one of FLUSHR, FLUSHW, or FLUSHRW as described above. The *bi_pri* member determines the priority band to be flushed.

I_SETSIG Requests that the STREAMS implementation send the SIGPOLL signal to the calling process when a particular event has occurred on the STREAM associated with *fildes*. I_SETSIG supports an asynchronous processing capability in STREAMS. The value of *arg* is a bitmask that specifies the events for which the process should be signaled. It is the bitwise-inclusive OR of any combination of the following constants:

- S_RDNORM A normal (priority band set to 0) message has arrived at the head of a STREAM head read queue. A signal shall be generated even if the message is of zero length.
- S_RDBAND A message with a non-zero priority band has arrived at the head of a STREAM head read queue. A signal shall be generated even if the message is of zero length.
- S_INPUT A message, other than a high-priority message, has arrived at the head of a STREAM head read queue. A signal shall be generated even if the message is of zero length.
- S_HIPRI A high-priority message is present on a STREAM head read queue. A signal shall be generated even if the message is of zero length.
- S_OUTPUT The write queue for normal data (priority band 0) just below the STREAM head is no longer full. This notifies the process that there is room on the queue for sending (or writing) normal data downstream.
- S_WRNORM Equivalent to S_OUTPUT.
- S_WRBAND The write queue for a non-zero priority band just below the STREAM head is no longer full. This notifies the process that there is room on the queue for sending (or writing) priority data downstream.
- S_MSG A STREAMS signal message that contains the SIGPOLL signal has reached the front of the STREAM head read queue.

S_ERROR	Notification of an error condition has reached the STREAM head.
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.
[EAGAIN]	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 *maxlen* member in the **ctlbuf** and **databuf** **strbuf** structures is set to the number of bytes of control information and/or data information, respectively, to retrieve. The *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:

RNORM	Byte-stream mode, the default.
RMSGD	Message-discard mode.
RMSGN	Message-nondiscard mode.

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 **ctlbuf 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_timout* is the number of seconds (–1=infinite, 0=use implementation-defined timeout interval, >0=as specified) an I_STR request shall wait for acknowledgement before timing out. *ic_len* is the number of bytes in the data argument, and *ic_dp* is a pointer to the data argument. The *ic_len* member has two uses: on input, it contains the length of the data argument passed in, and on return from the command, it contains the number of bytes being returned to the process (the buffer pointed to by *ic_dp* should be large enough to contain the maximum amount of data that any module or the driver in the STREAM can return).

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_timout* 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.

21342 [EBADF] The *arg* argument is not a valid, open file descriptor.

21343 [EINVAL] The *fildes* argument is not connected to a STREAM pipe.

21344 [ENXIO] Hangup received on *fildes*.

21345 I_RECVFD Retrieves the reference to an open file description from a message written to a
21346 STREAMS-based pipe using the I_SENDFD command, and allocates a new file
21347 descriptor in the calling process that refers to this open file description. The
21348 *arg* argument is a pointer to a **strrecvfd** data structure as defined in
21349 **<stropts.h>**.

21350 The *fd* member is a file descriptor. The *uid* and *gid* members are the effective
21351 user ID and effective group ID, respectively, of the sending process.

21352 If O_NONBLOCK is not set, I_RECVFD shall block until a message is present
21353 at the STREAM head. If O_NONBLOCK is set, I_RECVFD shall fail with *errno*
21354 set to [EAGAIN] if no message is present at the STREAM head.

21355 If the message at the STREAM head is a message sent by an I_SENDFD, a new
21356 file descriptor shall be allocated for the open file descriptor referenced in the
21357 message. The new file descriptor is placed in the *fd* member of the **strrecvfd**
21358 structure pointed to by *arg*.

21359 The *ioctl*( ) function with the I_RECVFD command shall fail if:

21360 [EAGAIN] A message is not present at the STREAM head read queue
21361 and the O_NONBLOCK flag is set.

21362 [EBADMSG] The message at the STREAM head read queue is not a
21363 message containing a passed file descriptor.

21364 [EMFILE] All file descriptors available to the process are currently
21365 open.

21366 [ENXIO] Hangup received on *fildes*.

21367 I_LIST Allows the process to list all the module names on the STREAM, up to and
21368 including the topmost driver name. If *arg* is a null pointer, the return value
21369 shall be the number of modules, including the driver, that are on the STREAM
21370 pointed to by *fildes*. This lets the process allocate enough space for the module
21371 names. Otherwise, it should point to a **str_list** structure.

21372 The *sl_nmods* member indicates the number of entries the process has
21373 allocated in the array. Upon return, the *sl_modlist* member of the **str_list**
21374 structure shall contain the list of module names, and the number of entries
21375 that have been filled into the *sl_modlist* array is found in the *sl_nmods* member
21376 (the number includes the number of modules including the driver). The return
21377 value from *ioctl*( ) shall be 0. The entries are filled in starting at the top of the
21378 STREAM and continuing downstream until either the end of the STREAM is
21379 reached, or the number of requested modules (*sl_nmods*) is satisfied.

21380 The *ioctl*( ) function with the I_LIST command shall fail if:

21381 [EINVAL] The *sl_nmods* member is less than 1.

21382 [EAGAIN] or [ENOSR]
21383 Unable to allocate buffers.

21384 I_ATMARK Allows the process to see if the message at the head of the STREAM head read
21385 queue is marked by some module downstream. The *arg* argument determines
21386 how the checking is done when there may be multiple marked messages on
21387 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

21478 multiplexer ID number (an identifier that may be used to disconnect the
21479 multiplexer; see I_PUNLINK) on success, and –1 on failure.

21480 The *ioctl*( ) function with the I_PLINK command shall fail if:

21481 [ENXIO] Hangup received on *fildes*.

21482 [ETIME] Timeout before acknowledgement message was received at
21483 STREAM head.

21484 [EAGAIN] or [ENOSR]
21485 Unable to allocate STREAMS storage to perform the
21486 I_PLINK.

21487 [EBADF] The *arg* argument is not a valid, open file descriptor.

21488 [EINVAL] The *fildes* argument does not support multiplexing; or *arg* is
21489 not a STREAM or is already connected downstream from a
21490 multiplexer; or the specified I_PLINK operation would
21491 connect the STREAM head in more than one place in the
21492 multiplexed STREAM.

21493 An I_PLINK can also fail while waiting for the multiplexing driver to
21494 acknowledge the request, if a message indicating an error or a hangup is
21495 received at the STREAM head of *fildes*. In addition, an error code can be
21496 returned in the positive or negative acknowledgement message. For these
21497 cases, I_PLINK shall fail with *errno* set to the value in the message.

21498 I_PUNLINK Disconnects the two STREAMs specified by *fildes* and *arg* from a persistent
21499 connection. The *fildes* argument is the file descriptor of the STREAM
21500 connected to the multiplexing driver. The *arg* argument is the multiplexer ID
21501 number that was returned by the I_PLINK *ioctl*( ) command when a STREAM
21502 was connected downstream from the multiplexing driver. If *arg* is
21503 MUXID_ALL, then all STREAMs which are persistent connections to *fildes*
21504 shall be disconnected. As in I_PLINK, this command requires the multiplexing
21505 driver to acknowledge the request.

21506 The *ioctl*( ) function with the I_PUNLINK command shall fail if:

21507 [ENXIO] Hangup received on *fildes*.

21508 [ETIME] Timeout before acknowledgement message was received at
21509 STREAM head.

21510 [EAGAIN] or [ENOSR]
21511 Unable to allocate buffers for the acknowledgement
21512 message.

21513 [EINVAL] Invalid multiplexer ID number.

21514 An I_PUNLINK can also fail while waiting for the multiplexing driver to
21515 acknowledge the request if a message indicating an error or a hangup is
21516 received at the STREAM head of *fildes*. In addition, an error code can be
21517 returned in the positive or negative acknowledgement message. For these
21518 cases, I_PUNLINK shall fail with *errno* set to the value in the message.

21519 **RETURN VALUE**

21520 Upon successful completion, *ioctl*( ) shall return a value other than –1 that depends upon the
21521 STREAMS device control function. Otherwise, it shall return –1 and set *errno* to indicate the
21522 error.

21523 **ERRORS**

21524 Under the following general conditions, *ioctl*( ) shall fail if:

21525 [EBADF] The *fildes* argument is not a valid open file descriptor.

21526 [EINTR] A signal was caught during the *ioctl*( ) operation.

21527 21528 [EINVAL] The STREAM or multiplexer referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexer.

21529 If an underlying device driver detects an error, then *ioctl*( ) shall fail if:

21530 [EINVAL] The *request* or *arg* argument is not valid for this device.

21531 [EIO] Some physical I/O error has occurred.

21532 21533 [ENOTTY] The file associated with the *fildes* argument is not a STREAMS device that accepts control functions.

21534 21535 [ENXIO] The *request* and *arg* arguments are valid for this device driver, but the service requested cannot be performed on this particular sub-device.

21536 21537 [ENODEV] The *fildes* argument refers to a valid STREAMS device, but the corresponding device driver does not support the *ioctl*( ) function.

21538 21539 If a STREAM is connected downstream from a multiplexer, any *ioctl*( ) command except I_UNLINK and I_PUNLINK shall set *errno* to [EINVAL].

21540 **EXAMPLES**

21541 None.

21542 **APPLICATION USAGE**

21543 The implementation-defined timeout interval for STREAMS has historically been 15 seconds.

21544 **RATIONALE**

21545 None.

21546 **FUTURE DIRECTIONS**

21547 The *ioctl*( ) function may be removed in a future version.

21548 **SEE ALSO**

21549 21550 Section 2.6 (on page 38), *close*( ), *fcntl*( ), *getmsg*( ), *open*( ), *pipe*( ), *poll*( ), *putmsg*( ), *read*( ), *sigaction*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stropts.h>**

21551 **CHANGE HISTORY**

21552 First released in Issue 4, Version 2.

21553 **Issue 5**

21554 Moved from X/OPEN UNIX extension to BASE.

21555 **Issue 6**

21556 21557 The Open Group Corrigendum U028/4 is applied, correcting text in the I_FDINSERT [EINVAL] case to refer to *ctlbuf*.

21558 This function is marked as part of the XSI STREAMS Option Group.

21559 The normative text is updated to avoid use of the term "must" for application requirements.

21560 **Issue 7**

21561 SD5-XSH-ERN-100 is applied, correcting the definition of the [ENOTTY] error condition.

21562 The *ioctl*( ) function is marked obsolescent.

21563 **NAME**

21564 isalnum, isalnum_l — test for an alphanumeric character

21565 **SYNOPSIS**

21566
```
#include <ctype.h>
```

21567
```
int isalnum(int c);
```

21568 CX
```
int isalnum_l(int c, locale_t locale);
```

21569 **DESCRIPTION**

21570 CX For *isalnum*( ): The functionality described on this reference page is aligned with the ISO C
21571 standard. Any conflict between the requirements described here and the ISO C standard is
21572 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21573 CX The *isalnum*( ) and *isalnum_l*( ) functions shall test whether *c* is a character of class **alpha** or
21574 CX **digit** in the current locale of the process, or in the locale represented by *locale*, respectively; see
21575 the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

21576 The *c* argument is an **int**, the value of which the application shall ensure is representable as an
21577 **unsigned char** or equal to the value of the macro EOF. If the argument has any other value, the
21578 behavior is undefined.

21579 **RETURN VALUE**

21580 CX The *isalnum*( ) and *isalnum_l*( ) functions shall return non-zero if *c* is an alphanumeric character;
21581 otherwise, they shall return 0.

21582 **ERRORS**

21583 The *isalnum_l*( ) function may fail if:

21584 CX [EINVAL] *locale* is not a valid locale object handle.

21585 **EXAMPLES**

21586 None.

21587 **APPLICATION USAGE**

21588 To ensure applications portability, especially across natural languages, only these functions and
21589 the functions in the reference pages listed in the SEE ALSO section should be used for character
21590 classification.

21591 **RATIONALE**

21592 None.

21593 **FUTURE DIRECTIONS**

21594 None.

21595 **SEE ALSO**

21596 *isalpha*( ), *isblank*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ),
21597 *isxdigit*( ), *setlocale*( ), *uselocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7,
21598 Locale, **<ctype.h>**, **<stdio.h>**

21599 **CHANGE HISTORY**

21600 First released in Issue 1. Derived from Issue 1 of the SVID.

21601 **Issue 6**

21602 The normative text is updated to avoid use of the term "must" for application requirements.

Issue 7

The *isalnum_l*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

21606 **NAME**

21607 isalpha, isalpha_l — test for an alphabetic character

21608 **SYNOPSIS**

21609
```
#include <ctype.h>
```

21610
```
int isalpha(int c);
```

21611 CX
```
int isalpha_l(int c, locale_t locale);
```

21612 **DESCRIPTION**

21613 CX For *isalpha*( ): The functionality described on this reference page is aligned with the ISO C
21614 standard. Any conflict between the requirements described here and the ISO C standard is
21615 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21616 CX The *isalpha*( ) and *isalpha_l*( ) functions shall test whether *c* is a character of class **alpha** in the
21617 CX current locale of the process, or in the locale represented by *locale*, respectively; see the Base
21618 Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

21619 The *c* argument is an **int**, the value of which the application shall ensure is representable as an
21620 **unsigned char** or equal to the value of the macro EOF. If the argument has any other value, the
21621 behavior is undefined.

21622 **RETURN VALUE**

21623 CX The *isalpha*( ) and *isalpha_l*( ) functions shall return non-zero if *c* is an alphabetic character;
21624 otherwise, they shall return 0.

21625 **ERRORS**

21626 The *isalpha_l*( ) function may fail if:

21627 CX [EINVAL] *locale* is not a valid locale object handle.

21628 **EXAMPLES**

21629 None.

21630 **APPLICATION USAGE**

21631 To ensure applications portability, especially across natural languages, only these functions and
21632 the functions in the reference pages listed in the SEE ALSO section should be used for character
21633 classification.

21634 **RATIONALE**

21635 None.

21636 **FUTURE DIRECTIONS**

21637 None.

21638 **SEE ALSO**

21639 *isalnum*( ), *isblank*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ),
21640 *isxdigit*( ), *setlocale*( ), *uselocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7,
21641 Locale, **<ctype.h>**, **<locale.h>**, **<stdio.h>**

21642 **CHANGE HISTORY**

21643 First released in Issue 1. Derived from Issue 1 of the SVID.

21644 **Issue 6**

21645 The normative text is updated to avoid use of the term "must" for application requirements.

Issue 7

The *isalpha_l*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

21649 **NAME**

21650 isascii — test for a 7-bit US-ASCII character

21651 **SYNOPSIS**

21652 OB XSI
```
#include <ctype.h>

int isascii(int c);
```

21654 **DESCRIPTION**

21655 The *isascii*( ) function shall test whether *c* is a 7-bit US-ASCII character code.

21656 The *isascii*( ) function is defined on all integer values.

21657 **RETURN VALUE**

21658 The *isascii*( ) function shall return non-zero if *c* is a 7-bit US-ASCII character code between 0 and
21659 octal 0177 inclusive; otherwise, it shall return 0.

21660 **ERRORS**

21661 No errors are defined.

21662 **EXAMPLES**

21663 None.

21664 **APPLICATION USAGE**

21665 The *isascii*( ) function cannot be used portably in a localized application.

21666 **RATIONALE**

21667 None.

21668 **FUTURE DIRECTIONS**

21669 The *isascii*( ) function may be removed in a future version.

21670 **SEE ALSO**

21671 The Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**

21672 **CHANGE HISTORY**

21673 First released in Issue 1. Derived from Issue 1 of the SVID.

21674 **Issue 7**

21675 The *isascii*( ) function is marked obsolescent.

21676 **NAME**

21677 isastream — test a file descriptor (**STREAMS**)

21678 **SYNOPSIS**

21679 OB XSR
```
#include <stropts.h>

int isastream(int fildes);
```

21680

21681 **DESCRIPTION**

21682 The *isastream*( ) function shall test whether *fildes*, an open file descriptor, is associated with a
21683 STREAMS-based file.

21684 **RETURN VALUE**

21685 Upon successful completion, *isastream*( ) shall return 1 if *fildes* refers to a STREAMS-based file
21686 and 0 if not. Otherwise, *isastream*( ) shall return –1 and set *errno* to indicate the error.

21687 **ERRORS**

21688 The *isastream*( ) function shall fail if:

21689 [EBADF] The *fildes* argument is not a valid open file descriptor.

21690 **EXAMPLES**

21691 None.

21692 **APPLICATION USAGE**

21693 None.

21694 **RATIONALE**

21695 None.

21696 **FUTURE DIRECTIONS**

21697 The *isastream*( ) function may be removed in a future version.

21698 **SEE ALSO**

21699 The Base Definitions volume of IEEE Std 1003.1-200x, **<stropts.h>**

21700 **CHANGE HISTORY**

21701 First released in Issue 4, Version 2.

21702 **Issue 5**

21703 Moved from X/OPEN UNIX extension to BASE.

21704 **Issue 7**

21705 The *isastream*( ) function is marked obsolescent.

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 file associated with the *fildes* argument is not 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 1003.1-200x, **<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.

Issue 7

SD5-XSH-ERN-100 is applied, correcting the definition of the [ENOTTY] error condition.

21742 **NAME**

21743 isblank, isblank_l — test for a blank character

21744 **SYNOPSIS**

21745
```
#include <ctype.h>
```

21746
```
int isblank(int c);
```

21747 CX
```
int isblank_l(int c, locale_t locale);
```

21748 **DESCRIPTION**

21749 CX For *isblank*( ): The functionality described on this reference page is aligned with the ISO C
21750 standard. Any conflict between the requirements described here and the ISO C standard is
21751 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21752 CX The *isblank*( ) and *isblank_l*( ) functions shall test whether *c* is a character of class **blank** in the
21753 CX current locale of the process, or in the locale respresented by *locale*, respectively; see the Base
21754 Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

21755 The *c* argument is a type **int**, the value of which the application shall ensure is a character
21756 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
21757 any other value, the behavior is undefined.

21758 **RETURN VALUE**

21759 CX The *isblank*( ) and *isblank_l*( ) functions shall return non-zero if *c* is a <blank>; otherwise, they
21760 shall return 0.

21761 **ERRORS**

21762 The *isblank_l*( ) function may fail if:

21763 CX [EINVAL] *locale* is not a valid locale object handle.

21764 **EXAMPLES**

21765 None.

21766 **APPLICATION USAGE**

21767 To ensure applications portability, especially across natural languages, only these functions and
21768 the functions in the reference pages listed in the SEE ALSO section should be used for character
21769 classification.

21770 **RATIONALE**

21771 None.

21772 **FUTURE DIRECTIONS**

21773 None.

21774 **SEE ALSO**

21775 *isalnum*( ), *isalpha*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ),
21776 *isxdigit*( ), *setlocale*( ), *uselocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7,
21777 Locale, **<ctype.h>**, **<locale.h>**

21778 **CHANGE HISTORY**

21779 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

21780 **Issue 7**

21781 The *isblank_l*( ) function is added from The Open Group Technical Standard, 2006, Extended API
21782 Set Part 4.

21783 **NAME**

21784 iscntrl, iscntrl_l — test for a control character

21785 **SYNOPSIS**

21786
```
#include <ctype.h>
```

21787
```
int iscntrl(int c);
```

21788 CX
```
int iscntrl_l(int c, locale_t locale);
```

21789 **DESCRIPTION**

21790 CX For *iscntrl*( ): The functionality described on this reference page is aligned with the ISO C
21791 standard. Any conflict between the requirements described here and the ISO C standard is
21792 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21793 CX The *iscntrl*( ) and *iscntrl_l*( ) functions shall test whether *c* is a character of class **cntrl** in the
21794 CX current locale of the process, or in the locale represented by *locale*, respectively; see the Base
21795 Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

21796 The *c* argument is a type **int**, the value of which the application shall ensure is a character
21797 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
21798 any other value, the behavior is undefined.

21799 **RETURN VALUE**

21800 CX The *iscntrl*( ) and *iscntrl_l*( ) functions shall return non-zero if *c* is a control character; otherwise,
21801 they shall return 0.

21802 **ERRORS**

21803 The *iscntrl_l*( ) function may fail if:

21804 CX [EINVAL] *locale* is not a valid locale object handle.

21805 **EXAMPLES**

21806 None.

21807 **APPLICATION USAGE**

21808 To ensure applications portability, especially across natural languages, only these functions and
21809 the functions in the reference pages listed in the SEE ALSO section should be used for character
21810 classification.

21811 **RATIONALE**

21812 None.

21813 **FUTURE DIRECTIONS**

21814 None.

21815 **SEE ALSO**

21816 *isalnum*( ), *isalpha*( ), *isblank*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ),
21817 *isupper*( ), *isxdigit*( ), *setlocale*( ), *uselocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
21818 Chapter 7, Locale, **<ctype.h>**, **<locale.h>**

21819 **CHANGE HISTORY**

21820 First released in Issue 1. Derived from Issue 1 of the SVID.

21821 **Issue 6**

21822 The normative text is updated to avoid use of the term "must" for application requirements.

Issue 7

The *iscntrl_l*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

NAME

isdigit, isdigit_l — test for a decimal digit

SYNOPSIS

```
#include <ctype.h>

int isdigit(int c);
CX int isdigit_l(int c, locale_t locale);
```

DESCRIPTION

CX For *isdigit*(): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

CX The *isdigit*() and *isdigit_l*() functions shall test whether *c* is a character of class **digit** in the CX current locale of the process, or in the locale represented by *locale*, respectively; see the Base Definitions volume of IEEE Std 1003.1-200x, 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

CX The *isdigit*() and *isdigit_l*() functions shall return non-zero if *c* is a decimal digit; otherwise, they shall return 0.

ERRORS

The *isdigit_l*() function may fail if:

CX [EINVAL] *locale* is not a valid locale object handle.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only these functions and the functions in the reference pages listed in the SEE ALSO section should be used for character classification.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalnum(), *isalpha*(), *isblank*(), *iscntrl*(), *isgraph*(), *islower*(), *isprint*(), *ispunct*(), *isspace*(), *isupper*(), *isxdigit*(), the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale, **<ctype.h>**, **<locale.h>**

CHANGE HISTORY

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

Issue 6

The normative text is updated to avoid use of the term "must" for application requirements.

Issue 7

The *isdigit_l*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

21869 **NAME**

21870 isfinite — test for finite value

21871 **SYNOPSIS**

21872
```
#include <math.h>
```

21873
```
int isfinite(real-floating x);
```

21874 **DESCRIPTION**

21875 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21876 conflict between the requirements described here and the ISO C standard is unintentional. This
21877 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21878 The *isfinite*( ) macro shall determine whether its argument has a finite value (zero, subnormal, or
21879 normal, and not infinite or NaN). First, an argument represented in a format wider than its
21880 semantic type is converted to its semantic type. Then determination is based on the type of the
21881 argument.

21882 **RETURN VALUE**

21883 The *isfinite*( ) macro shall return a non-zero value if and only if its argument has a finite value.

21884 **ERRORS**

21885 No errors are defined.

21886 **EXAMPLES**

21887 None.

21888 **APPLICATION USAGE**

21889 None.

21890 **RATIONALE**

21891 None.

21892 **FUTURE DIRECTIONS**

21893 None.

21894 **SEE ALSO**

21895 *fpclassify*( ), *isinf*( ), *isnan*( ), *isnormal*( ), *signbit*( ), the Base Definitions volume of
21896 IEEE Std 1003.1-200x **<math.h>**

21897 **CHANGE HISTORY**

21898 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

21899 **NAME**

21900 isgraph, isgraph_l — test for a visible character

21901 **SYNOPSIS**

21902
```
#include <ctype.h>
```

21903
```
int isgraph(int c);
```

21904 CX
```
int isgraph_l(int c, locale_t locale);
```

21905 **DESCRIPTION**

21906 CX For *isgraph*( ): The functionality described on this reference page is aligned with the ISO C
21907 standard. Any conflict between the requirements described here and the ISO C standard is
21908 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21909 CX The *isgraph*( ) and *isgraph_l*( ) functions shall test whether *c* is a character of class **graph** in the
21910 CX current locale of the process, or in the locale represented by *locale*, respectively; see the Base
21911 Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

21912 The *c* argument is an **int**, the value of which the application shall ensure is a character
21913 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
21914 any other value, the behavior is undefined.

21915 **RETURN VALUE**

21916 CX The *isgraph*( ) and *isgraph_l*( ) functions shall return non-zero if *c* is a character with a visible
21917 representation; otherwise, they shall return 0.

21918 **ERRORS**

21919 The *isgraph_l*( ) function may fail if:

21920 CX [EINVAL] *locale* is not a valid locale object handle.

21921 **EXAMPLES**

21922 None.

21923 **APPLICATION USAGE**

21924 To ensure applications portability, especially across natural languages, only these functions and
21925 the functions in the reference pages listed in the SEE ALSO section should be used for character
21926 classification.

21927 **RATIONALE**

21928 None.

21929 **FUTURE DIRECTIONS**

21930 None.

21931 **SEE ALSO**

21932 *isalnum*( ), *isalpha*( ), *isblank*( ), *iscntrl*( ), *isdigit*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ),
21933 *isxdigit*( ), *setlocale*( ), *uselocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7,
21934 Locale, **<ctype.h>**, **<locale.h>**

21935 **CHANGE HISTORY**

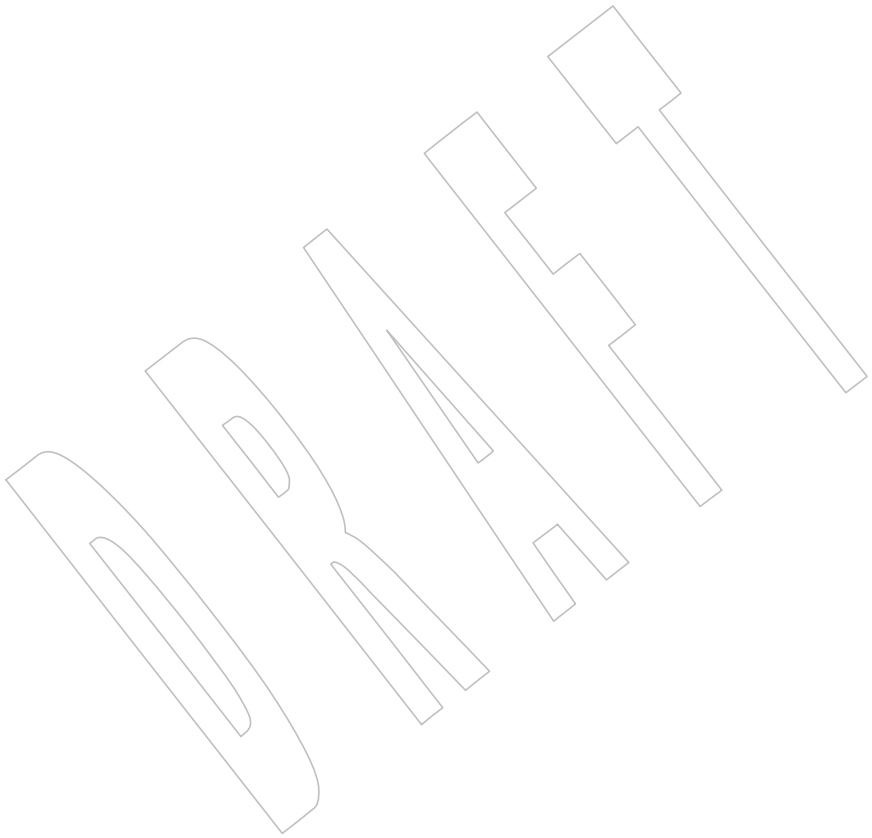
21936 First released in Issue 1. Derived from Issue 1 of the SVID.

21937 **Issue 6**

21938 The normative text is updated to avoid use of the term "must" for application requirements.

Issue 7

The *isgraph_l*( ) function is added from The Open Group Technical Standard, 2006, Extended API



21942 **NAME**

21943 isgreater — test if x greater than y

21944 **SYNOPSIS**

21945
```
#include <math.h>
```

21946
```
int isgreater(real-floating x, real-floating y);
```

21947 **DESCRIPTION**

21948 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21949 conflict between the requirements described here and the ISO C standard is unintentional. This
21950 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

21951 The *isgreater*( ) macro shall determine whether its first argument is greater than its second
21952 argument. The value of *isgreater*(*x*, *y*) shall be equal to (*x*) > (*y*); however, unlike (*x*) > (*y*),
21953 *isgreater*(*x*, *y*) shall not raise the invalid floating-point exception when *x* and *y* are unordered.

21954 **RETURN VALUE**

21955 Upon successful completion, the *isgreater*( ) macro shall return the value of (*x*) > (*y*).

21956 If *x* or *y* is NaN, 0 shall be returned.

21957 **ERRORS**

21958 No errors are defined.

21959 **EXAMPLES**

21960 None.

21961 **APPLICATION USAGE**

21962 The relational and equality operators support the usual mathematical relationships between
21963 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
21964 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
21965 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
21966 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
21967 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
21968 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
21969 indicates that the argument shall be an expression of **real-floating** type.

21970 **RATIONALE**

21971 None.

21972 **FUTURE DIRECTIONS**

21973 None.

21974 **SEE ALSO**

21975 *isgreaterequal*( ), *isless*( ), *islessequal*( ), *islessgreater*( ), *isunordered*( ), the Base Definitions volume of
21976 IEEE Std 1003.1-200x **<math.h>**

21977 **CHANGE HISTORY**

21978 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

NAME

isgreaterequal — test if x is greater than or equal to y

SYNOPSIS

```
#include <math.h>

int isgreaterequal(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 1003.1-200x defers to the ISO C standard.

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.

RETURN VALUE

Upon successful completion, the *isgreaterequal*( ) 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( ), *isless*( ), *islessequal*( ), *islessgreater*( ), *isunordered*( ), the Base Definitions volume of IEEE Std 1003.1-200x **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

22017 **NAME**

22018 isinf — test for infinity

22019 **SYNOPSIS**

22020
```
#include <math.h>
```

22021
```
int isinf(real-floating x);
```

22022 **DESCRIPTION**

22023 CX The functionality described on this reference page is aligned with the ISO C standard. Any
22024 conflict between the requirements described here and the ISO C standard is unintentional. This
22025 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22026 The *isinf*( ) macro shall determine whether its argument value is an infinity (positive or
22027 negative). First, an argument represented in a format wider than its semantic type is converted
22028 to its semantic type. Then determination is based on the type of the argument.

22029 **RETURN VALUE**

22030 The *isinf*( ) macro shall return a non-zero value if and only if its argument has an infinite value.

22031 **ERRORS**

22032 No errors are defined.

22033 **EXAMPLES**

22034 None.

22035 **APPLICATION USAGE**

22036 None.

22037 **RATIONALE**

22038 None.

22039 **FUTURE DIRECTIONS**

22040 None.

22041 **SEE ALSO**

22042 *fpclassify*( ), *isfinite*( ), *isnan*( ), *isnormal*( ), *signbit*( ), the Base Definitions volume of
22043 IEEE Std 1003.1-200x **<math.h>**

22044 **CHANGE HISTORY**

22045 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

22046 **NAME**

22047 isless — test if x is less than y

22048 **SYNOPSIS**

22049
```
#include <math.h>
```

22050
```
int isless(real-floating x, real-floating y);
```

22051 **DESCRIPTION**

22052 CX The functionality described on this reference page is aligned with the ISO C standard. Any
22053 conflict between the requirements described here and the ISO C standard is unintentional. This
22054 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22055 The *isless*( ) macro shall determine whether its first argument is less than its second argument.
22056 The value of *isless*(*x*, *y*) shall be equal to (*x*) < (*y*); however, unlike (*x*) < (*y*), *isless*(*x*, *y*) shall not
22057 raise the invalid floating-point exception when *x* and *y* are unordered.

22058 **RETURN VALUE**

22059 Upon successful completion, the *isless*( ) macro shall return the value of (*x*) < (*y*).

22060 If *x* or *y* is NaN, 0 shall be returned.

22061 **ERRORS**

22062 No errors are defined.

22063 **EXAMPLES**

22064 None.

22065 **APPLICATION USAGE**

22066 The relational and equality operators support the usual mathematical relationships between
22067 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
22068 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
22069 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
22070 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
22071 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
22072 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
22073 indicates that the argument shall be an expression of **real-floating** type.

22074 **RATIONALE**

22075 None.

22076 **FUTURE DIRECTIONS**

22077 None.

22078 **SEE ALSO**

22079 *isgreater*( ), *isgreaterequal*( ), *islessequal*( ), *islessgreater*( ), *isunordered*( ), the Base Definitions volume
22080 of IEEE Std 1003.1-200x, **<math.h>**

22081 **CHANGE HISTORY**

22082 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

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 1003.1-200x 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

22121 **NAME**

22122 islessgreater — test if x is less than or greater than y

22123 **SYNOPSIS**

22124
```
#include <math.h>
```

22125
```
int islessgreater(real-floating x, real-floating y);
```

22126 **DESCRIPTION**

22127 CX The functionality described on this reference page is aligned with the ISO C standard. Any
22128 conflict between the requirements described here and the ISO C standard is unintentional. This
22129 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22130 The *islessgreater*( ) macro shall determine whether its first argument is less than or greater than
22131 its second argument. The *islessgreater*(*x*, *y*) macro is similar to (*x*) < (*y*) || (*x*) > (*y*); however,
22132 *islessgreater*(*x*, *y*) shall not raise the invalid floating-point exception when *x* and *y* are unordered
22133 (nor shall it evaluate *x* and *y* twice).

22134 **RETURN VALUE**

22135 Upon successful completion, the *islessgreater*( ) macro shall return the value of
22136 (*x*) < (*y*) || (*x*) > (*y*).

22137 If *x* or *y* is NaN, 0 shall be returned.

22138 **ERRORS**

22139 No errors are defined.

22140 **EXAMPLES**

22141 None.

22142 **APPLICATION USAGE**

22143 The relational and equality operators support the usual mathematical relationships between
22144 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
22145 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
22146 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
22147 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
22148 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
22149 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
22150 indicates that the argument shall be an expression of **real-floating** type.

22151 **RATIONALE**

22152 None.

22153 **FUTURE DIRECTIONS**

22154 None.

22155 **SEE ALSO**

22156 *isgreater*( ), *isgreaterequal*( ), *isless*( ), *islessequal*( ), *isunordered*( ), the Base Definitions volume of
22157 IEEE Std 1003.1-200x **<math.h>**

22158 **CHANGE HISTORY**

22159 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

NAME

islower, islower_l — test for a lowercase letter

SYNOPSIS

```
#include <ctype.h>

int islower(int c);
CX  int islower_l(int c, locale_t locale);
```

DESCRIPTION

CX For *islower*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

CX The *islower*( ) and *islower_l*( ) functions shall test whether *c* is a character of class **lower** in the CX current locale of the process, or in the locale represented by *locale*, respectively; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

The *c* argument is an **int** **5Es**

```
            keystr[len++] = c;
        }
    ...
    /* Example 2 -- using islower_l() */
    #include <ctype.h>
    #include <stdlib.h>
    #include <locale.h>
    ...
    char *keystr;
    int elementlen, len;
    char c;
    ...
    locale_t loc = newlocale (LC_ALL_MASK, "", (locale_t) 0);
    ...
    len = 0;
    while (len < elementlen) {
        c = (char) (rand() % 256);
    ...
        if (islower_l(c, loc))
            keystr[len++] = c;
        }
    ...
```

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only these functions and the functions in the reference pages listed in the SEE ALSO section should be used for character classification.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalnum( ), *isalpha( )*, *isblank( )*, *iscntrl( )*, *isdigit( )*, *isgraph( )*, *isprint( )*, *ispunct( )*, *isspace( )*, *isupper( )*, *isxdigit( )*, *setlocale( )*, *uselocale( )*, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale, **<ctype.h>**, **<locale.h>**

CHANGE HISTORY

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

Issue 6

The normative text is updated to avoid use of the term "must" for application requirements.

An example is added.

Issue 7

The *islower_l( )* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

22246 **NAME**

22247 isnan — test for a NaN

22248 **SYNOPSIS**

22249
```
#include <math.h>
```

22250
```
int isnan(real-floating x);
```

22251 **DESCRIPTION**

22252 CX The functionality described on this reference page is aligned with the ISO C standard. Any
22253 conflict between the requirements described here and the ISO C standard is unintentional. This
22254 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22255 The *isnan*( ) macro shall determine whether its argument value is a NaN. First, an argument
22256 represented in a format wider than its semantic type is converted to its semantic type. Then
22257 determination is based on the type of the argument.

22258 **RETURN VALUE**

22259 The *isnan*( ) macro shall return a non-zero value if and only if its argument has a NaN value.

22260 **ERRORS**

22261 No errors are defined.

22262 **EXAMPLES**

22263 None.

22264 **APPLICATION USAGE**

22265 None.

22266 **RATIONALE**

22267 None.

22268 **FUTURE DIRECTIONS**

22269 None.

22270 **SEE ALSO**

22271 *fpclassify*( ), *isfinite*( ), *isinf*( ), *isnormal*( ), *signbit*( ), the Base Definitions volume of
22272 IEEE Std 1003.1-200x, **<math.h>**

22273 **CHANGE HISTORY**

22274 First released in Issue 3.

22275 **Issue 5**

22276 The DESCRIPTION is updated to indicate the return value when NaN is not supported. This
22277 text was previously published in the APPLICATION USAGE section.

22278 **Issue 6**

22279 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

CX The functionality described on this reference page is aligned with the ISO C standard. Any

22311 **NAME**

22312 isprint, isprint_l — test for a printable character

22313 **SYNOPSIS**

22314
```
#include <ctype.h>
```

22315
```
int isprint(int c);
```

22316 CX
```
int isprint_l(int c, locale_t locale);
```

22317 **DESCRIPTION**

22318 CX For *isprint*( ): The functionality described on this reference page is aligned with the ISO C
22319 standard. Any conflict between the requirements described here and the ISO C standard is
22320 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22321 CX The *isprint*( ) and *isprint_l*( ) functions shall test whether *c* is a character of class **print** in the
22322 CX current locale of the process, or in the locale represented by *locale*, respectively; see the Base
22323 Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

22324 The *c* argument is an **int**, the value of which the application shall ensure is a character
22325 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
22326 any other value, the behavior is undefined.

22327 **RETURN VALUE**

22328 CX The *isprint*( ) and *isprint_l*( ) functions shall return non-zero if *c* is a printable character;
22329 otherwise, they shall return 0.

22330 **ERRORS**

22331 The *isprint_l*( ) function may fail if:

22332 CX [EINVAL] *locale* is not a valid locale object handle.

22333 **EXAMPLES**

22334 None.

22335 **APPLICATION USAGE**

22336 To ensure applications portability, especially across natural languages, only these functions and
22337 the functions in the reference pages listed in the SEE ALSO section should be used for character
22338 classification.

22339 **RATIONALE**

22340 None.

22341 **FUTURE DIRECTIONS**

22342 None.

22343 **SEE ALSO**

22344 *isalnum*( ), *isalpha*( ), *isblank*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *ispunct*( ), *isspace*( ), *isupper*( ),
22345 *isxdigit*( ), *setlocale*( ), *uselocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7,
22346 Locale, **<ctype.h>**, **<locale.h>**

22347 **CHANGE HISTORY**

22348 First released in Issue 1. Derived from Issue 1 of the SVID.

22349 **Issue 6**

22350 The normative text is updated to avoid use of the term "must" for application requirements.

Issue 7

The *isprint_l*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

22354 **NAME**

22355 ispunct, ispunct_l — test for a punctuation character

22356 **SYNOPSIS**

22357
```
#include <ctype.h>
```

22358
```
int ispunct(int c);
```

22359 CX
```
int ispunct_l(int c, locale_t locale);
```

22360 **DESCRIPTION**

22361 CX For *ispunct*( ): The functionality described on this reference page is aligned with the ISO C
22362 standard. Any conflict between the requirements described here and the ISO C standard is
22363 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22364 CX The *ispunct*( ) and *ispunct_l*( ) functions shall test whether *c* is a character of class **punct** in the
22365 CX current locale of the process, or in the locale represented by *locale*, respectively; see the Base
22366 Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

22367 The *c* argument is an **int**, the value of which the application shall ensure is a character
22368 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
22369 any other value, the behavior is undefined.

22370 **RETURN VALUE**

22371 CX The *ispunct*( ) and *ispunct_l*( ) functions shall return non-zero if *c* is a punctuation character;
22372 otherwise, they shall return 0.

22373 **ERRORS**

22374 The *ispunct_l*( ) function may fail if:

22375 CX [EINVAL] *locale* is not a valid locale object handle.

22376 **EXAMPLES**

22377 None.

22378 **APPLICATION USAGE**

22379 To ensure applications portability, especially across natural languages, only these functions and
22380 the functions in the reference pages listed in the SEE ALSO section should be used for character
22381 classification.

22382 **RATIONALE**

22383 None.

22384 **FUTURE DIRECTIONS**

22385 None.

22386 **SEE ALSO**

22387 *isalnum*( ), *isalpha*( ), *isblank*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *isspace*( ), *isupper*( ),
22388 *isxdigit*( ), *setlocale*( ), *uselocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7,
22389 Locale, **<ctype.h>**, **<locale.h>**

22390 **CHANGE HISTORY**

22391 First released in Issue 1. Derived from Issue 1 of the SVID.

22392 **Issue 6**

22393 The normative text is updated to avoid use of the term "must" for application requirements.

22394 **Issue 7**

22395 The *ispunct_l*( ) function is added from The Open Group Technical Standard, 2006, Extended API
22396 Set Part 4.

22397 **NAME**

22398 isspace, isspace_l — test for a white-space character

22399 **SYNOPSIS**

22400
```
#include <ctype.h>
```

22401
```
int isspace(int c);
```

22402 CX
```
int isspace_l(int c, locale_t locale);
```

22403 **DESCRIPTION**

22404 CX For *isspace*( ): The functionality described on this reference page is aligned with the ISO C
22405 standard. Any conflict between the requirements described here and the ISO C standard is
22406 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22407 CX The *isspace*( ) and *isspace_l*( ) functions shall test whether *c* is a character of class **space** in the
22408 CX current locale of the process, or in the locale represented by *locale*, respectively; see the Base
22409 Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

22410 The *c* argument is an **int**, the value of which the application shall ensure is a character
22411 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
22412 any other value, the behavior is undefined.

22413 **RETURN VALUE**

22414 CX The *isspace*( ) and *isspace_l*( ) functions shall return non-zero if *c* is a white-space character;
22415 otherwise, they shall return 0.

22416 **ERRORS**

22417 The *isspace_l*( ) function may fail if:

22418 CX [EINVAL] *locale* is not a valid locale object handle.

22419 **EXAMPLES**

22420 None.

22421 **APPLICATION USAGE**

22422 To ensure applications portability, especially across natural languages, only these functions and
22423 the functions in the reference pages listed in the SEE ALSO section should be used for character
22424 classification.

22425 **RATIONALE**

22426 None.

22427 **FUTURE DIRECTIONS**

22428 None.

22429 **SEE ALSO**

22430 *isalnum*( ), *isalpha*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isupper*( ),
22431 *isxdigit*( ), *setlocale*( ), *uselocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7,
22432 Locale, **<ctype.h>**, **<locale.h>**

22433 **CHANGE HISTORY**

22434 First released in Issue 1. Derived from Issue 1 of the SVID.

22435 **Issue 6**

22436 The normative text is updated to avoid use of the term "must" for application requirements.

Issue 7

The *isspace_l*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

22440 **NAME**

22441 isunordered — test if arguments are unordered

22442 **SYNOPSIS**

22443
```
#include <math.h>
```

22444
```
int isunordered(real-floating x, real-floating y);
```

22445 **DESCRIPTION**

22446 CX The functionality described on this reference page is aligned with the ISO C standard. Any
22447 conflict between the requirements described here and the ISO C standard is unintentional. This
22448 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22449 The *isunordered*( ) macro shall determine whether its arguments are unordered.

22450 **RETURN VALUE**

22451 Upon successful completion, the *isunordered*( ) macro shall return 1 if its arguments are
22452 unordered, and 0 otherwise.

22453 If *x* or *y* is NaN, 1 shall be returned.

22454 **ERRORS**

22455 No errors are defined.

22456 **EXAMPLES**

22457 None.

22458 **APPLICATION USAGE**

22459 The relational and equality operators support the usual mathematical relationships between
22460 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
22461 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
22462 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
22463 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
22464 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
22465 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
22466 indicates that the argument shall be an expression of **real-floating** type.

22467 **RATIONALE**

22468 None.

22469 **FUTURE DIRECTIONS**

22470 None.

22471 **SEE ALSO**

22472 *isgreater*( ), *isgreaterequal*( ), *isless*( ), *islessequal*( ), *islessgreater*( ), the Base Definitions volume of
22473 IEEE Std 1003.1-200x, **<math.h>**

22474 **CHANGE HISTORY**

22475 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

22476 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/50 is applied, correcting the RETURN
22477 VALUE section when *x* or *y* is NaN.

22478 **NAME**

22479 isupper, isupper_l — test for an uppercase letter

22480 **SYNOPSIS**

22481
```
#include <ctype.h>
```

22482
```
int isupper(int c);
```

22483 CX
```
int isupper_l(int c, locale_t locale);
```

22484 **DESCRIPTION**

22485 CX For *isupper*( ): The functionality described on this reference page is aligned with the ISO C
22486 standard. Any conflict between the requirements described here and the ISO C standard is
22487 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22488 CX The *isupper*( ) and *isupper_l*( ) functions shall test whether *c* is a character of class **upper** in the
22489 CX current locale of the process, or in the locale represented by *locale*, respectively; see the Base
22490 Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

22491 The *c* argument is an **int**, the value of which the application shall ensure is a character
22492 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
22493 any other value, the behavior is undefined.

22494 **RETURN VALUE**

22495 CX The *isupper*( ) and *isupper_l*( ) functions shall return non-zero if *c* is an uppercase letter;
22496 otherwise, they shall return 0.

22497 **ERRORS**

22498 The *isupper_l*( ) function may fail if:

22499 CX [EINVAL] *locale* is not a valid locale object handle.

22500 **EXAMPLES**

22501 None.

22502 **APPLICATION USAGE**

22503 To ensure applications portability, especially across natural languages, only these functions and
22504 the functions in the reference pages listed in the SEE ALSO section should be used for character
22505 classification.

22506 **RATIONALE**

22507 None.

22508 **FUTURE DIRECTIONS**

22509 None.

22510 **SEE ALSO**

22511 *isalnum*( ), *isalpha*( ), *isblank*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ),
22512 *isxdigit*( ), *setlocale*( ), *uselocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7,
22513 Locale, **<ctype.h>**, **<locale.h>**

22514 **CHANGE HISTORY**

22515 First released in Issue 1. Derived from Issue 1 of the SVID.

22516 **Issue 6**

22517 The normative text is updated to avoid use of the term "must" for application requirements.

Issue 7

The *isupper_l*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

22521 **NAME**

22522 iswalnum, iswalnum_l — test for an alphanumeric wide-character code

22523 **SYNOPSIS**

22524
```
#include <wctype.h>
```

22525
```
int iswalnum(wint_t wc);
```

22526 CX
```
int iswalnum_l(wint_t wc, locale_t locale);
```

22527 **DESCRIPTION**

22528 CX For *iswalnum*( ): The functionality described on this reference page is aligned with the ISO C
22529 standard. Any conflict between the requirements described here and the ISO C standard is
22530 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22531 CX The *iswalnum*( ) and *iswalnum_l*( ) functions shall test whether *wc* is a wide-character code
22532 CX representing a character of class **alpha** or **digit** in the current locale of the process, or in the
22533 locale represented by *locale*, respectively; see the Base Definitions volume of
22534 IEEE Std 1003.1-200x, Chapter 7, Locale.

22535 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
22536 code corresponding to a valid character in the current locale, or equal to the value of the macro
22537 WEOF. If the argument has any other value, the behavior is undefined.

22538 **RETURN VALUE**

22539 CX The *iswalnum*( ) and *iswalnum_l*( ) functions shall return non-zero if *wc* is an alphanumeric
22540 wide-character code; otherwise, they shall return 0.

22541 **ERRORS**

22542 The *iswalnum_l*( ) function may fail if:

22543 CX [EINVAL] *locale* is not a valid locale object handle.

22544 **EXAMPLES**

22545 None.

22546 **APPLICATION USAGE**

22547 To ensure applications portability, especially across natural languages, only these functions and
22548 the functions in the reference pages listed in the SEE ALSO section should be used for character
22549 classification.

22550 **RATIONALE**

22551 None.

22552 **FUTURE DIRECTIONS**

22553 None.

22554 **SEE ALSO**

22555 *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ),
22556 *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), *uselocale*( ), the Base Definitions volume of
22557 IEEE Std 1003.1-200x, Chapter 7, Locale, **<locale.h>**, **<stdio.h>**, **<wctype.h>**

22558 **CHANGE HISTORY**

22559 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 normative text is updated to avoid use of the term "must" for application requirements.

Issue 7

The *iswalnum_l*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

22570 **NAME**

22571 iswalpha, iswalpha_l — test for an alphabetic wide-character code

22572 **SYNOPSIS**

22573
```
#include <wctype.h>
```

22574
```
int iswalpha(wint_t wc);
```

22575 CX
```
int iswalpha_l(wint_t wc, locale_t locale);
```

22576 **DESCRIPTION**

22577 CX For *iswalpha*( ): The functionality described on this reference page is aligned with the ISO C
22578 standard. Any conflict between the requirements described here and the ISO C standard is
22579 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22580 CX The *iswalpha*( ) and *iswalpha_l*( ) functions shall test whether *wc* is a wide-character code
22581 CX representing a character of class **alpha** in the current locale of the process, or in the locale
22582 represented by *locale*, respectively; see the Base Definitions volume of IEEE Std 1003.1-200x,
22583 Chapter 7, Locale.

22584 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
22585 code corresponding to a valid character in the current locale, or equal to the value of the macro
22586 WEOF. If the argument has any other value, the behavior is undefined.

22587 **RETURN VALUE**

22588 CX The *iswalpha*( ) and *iswalpha_l*( ) functions shall return non-zero if *wc* is an alphabetic wide-
22589 character code; otherwise, they shall return 0.

22590 **ERRORS**

22591 The *iswalpha_l*( ) function may fail if:

22592 CX [EINVAL] *locale* is not a valid locale object handle.

22593 **EXAMPLES**

22594 None.

22595 **APPLICATION USAGE**

22596 To ensure applications portability, especially across natural languages, only these functions and
22597 the functions in the reference pages listed in the SEE ALSO section should be used for character
22598 classification.

22599 **RATIONALE**

22600 None.

22601 **FUTURE DIRECTIONS**

22602 None.

22603 **SEE ALSO**

22604 *iswalnum*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ),
22605 *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), *uselocale*( ), the Base Definitions volume of
22606 IEEE Std 1003.1-200x, Chapter 7, Locale, **<locale.h>**, **<stdio.h>**, **<wctype.h>**

22607 **CHANGE HISTORY**

22608 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 normative text is updated to avoid use of the term "must" for application requirements.

Issue 7

The *iswalpha_l*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

22619 **NAME**

22620 iswblank, iswblank_l — test for a blank wide-character code

22621 **SYNOPSIS**

22622
```
#include <wctype.h>
```

22623
```
int iswblank(wint_t wc);
```

22624 CX
```
int iswblank_l(wint_t wc, locale_t locale);
```

22625 **DESCRIPTION**

22626 CX For *iswblank*( ): The functionality described on this reference page is aligned with the ISO C
22627 standard. Any conflict between the requirements described here and the ISO C standard is
22628 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22629 CX The *iswblank*( ) and *iswblank*( ) functions shall test whether *wc* is a wide-character code
22630 CX representing a character of class **blank** in the current locale of the process, or in the locale
22631 represented by *locale*, respectively; see the Base Definitions volume of IEEE Std 1003.1-200x,
22632 Chapter 7, Locale.

22633 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
22634 code corresponding to a valid character in the current locale, or equal to the value of the macro
22635 WEOF. If the argument has any other value, the behavior is undefined.

22636 **RETURN VALUE**

22637 CX The *iswblank*( ) and *iswblank_l*( ) functions shall return non-zero if *wc* is a blank wide-character
22638 code; otherwise, they shall return 0.

22639 **ERRORS**

22640 The *iswblank_l*( ) function may fail if:

22641 CX [EINVAL] *locale* is not a valid locale object handle.

22642 **EXAMPLES**

22643 None.

22644 **APPLICATION USAGE**

22645 To ensure applications portability, especially across natural languages, only these functions and
22646 the functions in the reference pages listed in the SEE ALSO section should be used for character
22647 classification.

22648 **RATIONALE**

22649 None.

22650 **FUTURE DIRECTIONS**

22651 None.

22652 **SEE ALSO**

22653 *iswalnum*( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ),
22654 *iswpunct*( ), *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), *uselocale*( ), the Base Definitions volume
22655 of IEEE Std 1003.1-200x, Chapter 7, Locale, **<locale.h>**, **<stdio.h>**, **<wctype.h>**

22656 **CHANGE HISTORY**

22657 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

Issue 7

The *iswblank_l*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

22661 **NAME**

22662 iswcntrl, iswcntrl_l — test for a control wide-character code

22663 **SYNOPSIS**

22664
```
#include <wctype.h>
```

22665
```
int iswcntrl(wint_t wc);
```

22666 CX
```
int iswcntrl_l(wint_t wc, locale_t locale);
```

22667 **DESCRIPTION**

22668 CX For *iswcntrl*( ): The functionality described on this reference page is aligned with the ISO C
22669 standard. Any conflict between the requirements described here and the ISO C standard is
22670 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22671 CX The *iswcntrl*( ) and *iswcntrl_l*( ) functions shall test whether *wc* is a wide-character code
22672 CX representing a character of class **cntrl** in the current locale of the process, or in the locale
22673 represented by *locale*, respectively; see the Base Definitions volume of IEEE Std 1003.1-200x,
22674 Chapter 7, Locale.

22675 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
22676 code corresponding to a valid character in the current locale, or equal to the value of the macro
22677 WEOF. If the argument has any other value, the behavior is undefined.

22678 **RETURN VALUE**

22679 CX The *iswcntrl*( ) and *iswcntrl_l*( ) functions shall return non-zero if *wc* is a control wide-character
22680 code; otherwise, they shall return 0.

22681 **ERRORS**

22682 The *iswcntrl_l*( ) function may fail if:

22683 CX [EINVAL] *locale* is not a valid locale object handle.

22684 **EXAMPLES**

22685 None.

22686 **APPLICATION USAGE**

22687 To ensure applications portability, especially across natural languages, only these functions and
22688 the functions in the reference pages listed in the SEE ALSO section should be used for character
22689 classification.

22690 **RATIONALE**

22691 None.

22692 **FUTURE DIRECTIONS**

22693 None.

22694 **SEE ALSO**

22695 *iswalnum*( ), *iswalpha*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ),
22696 *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), *uselocale*( ), the Base Definitions volume of
22697 IEEE Std 1003.1-200x, Chapter 7, Locale, **<locale.h>**, **<wctype.h>**

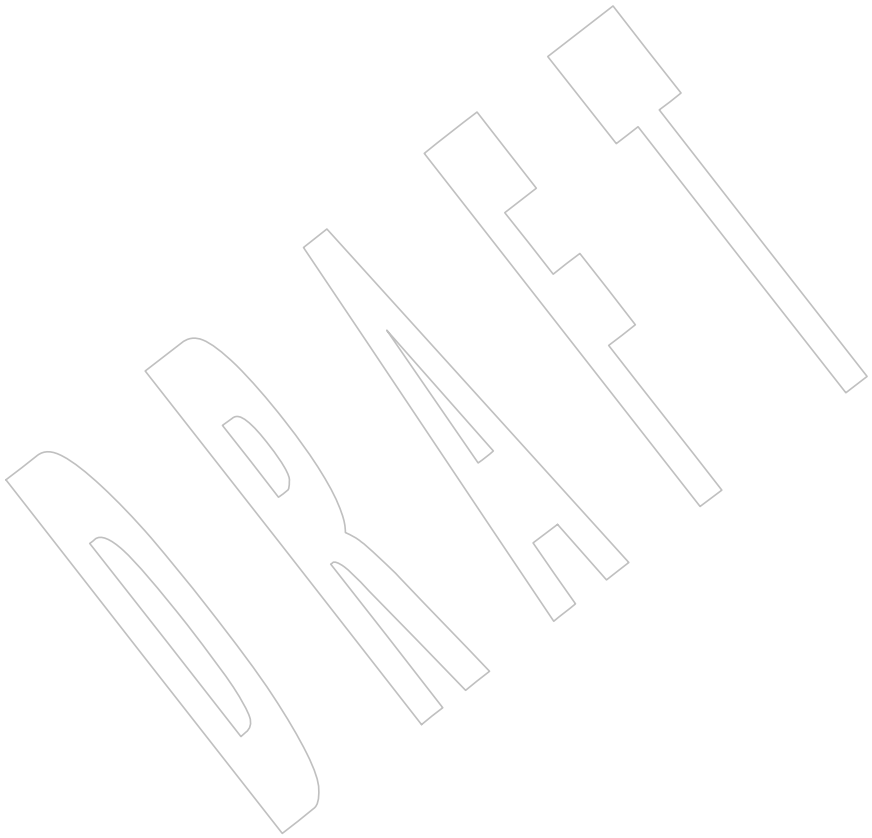
22698 **CHANGE HISTORY**

22699 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 n



NAME

iswctype, iswctype_l — test character for a specified class

SYNOPSIS

```
#include <wctype.h>

int iswctype(wint_t wc, wctype_t charclass);
CX int iswctype_l(wint_t wc, wctype_t charclass,
    locale_t locale);
```

DESCRIPTION

CX For *iswctype*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

CX The *iswctype*( ) and *iswctype_l*( ) functions shall determine whether the wide-character code *wc* CX has the character class *charclass*, returning true or false. The *iswctype*( ) and *iswctype_l*( ) functions are defined on WEOF and wide-character codes corresponding to the valid character CX encodings in the current locale, or in the locale represented by *locale*, respectively. 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

CX The *iswctype*( ) and *iswctype_l*( ) functions shall return non-zero (true) if and only if *wc* has the CX property described by *charclass*. If *charclass* is 0, these functions shall return 0.

ERRORS

The *iswctype_l*( ) function may fail if:

CX [EINVAL] *locale* is not a valid locale object handle.

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"))
iswalnum_l(wc, locale)  iswctype_l(wc, wctype("alnum"), locale)
iswalpha(wc)            iswctype(wc, wctype("alpha"))
iswalpha_l(wc, locale)  iswctype_l(wc, wctype("alpha"), locale)
```

```
iswblank(wc)            iswctype(wc, wctype("blank"))
iswblank_l(wc, locale)  iswctype_l(wc, wctype("blank"), locale)
iswcntrl(wc)            iswctype(wc, wctype("cntrl"))
iswcntrl_l(wc, locale)  iswctype_l(wc, wctype("cntrl"), locale)
iswdigit(wc)            iswctype(wc, wctype("digit"))
iswdigit_l(wc, locale)  iswctype_l(wc, wctype("digit"), locale)
iswgraph(wc)            iswctype(wc, wctype("graph"))
iswgraph_l(wc, locale)  iswctype_l(wc, wctype("graph"), locale)
iswlower(wc)            iswctype(wc, wctype("lower"))
iswlower_l(wc, locale)  iswctype_l(wc, wctype("lower"), locale)
iswprint(wc)            iswctype(wc, wctype("print"))
iswprint_l(wc, locale)  iswctype_l(wc, wctype("print"), locale)
iswpunct(wc)            iswctype(wc, wctype("punct"))
iswpunct_l(wc, locale)  iswctype_l(wc, wctype("punct"), locale)
iswspace(wc)            iswctype(wc, wctype("space"))
iswspace_l(wc, locale)  iswctype_l(wc, wctype("space"), locale)
iswupper(wc)            iswctype(wc, wctype("upper"))
iswupper_l(wc, locale)  iswctype_l(wc, wctype("upper"), locale)
iswxdigit(wc)           iswctype(wc, wctype("xdigit"))
iswxdigit_l(wc, locale) iswctype_l(wc, wctype("xdigit"), locale)
```

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswalnum( ), *iswalpha( )*, *iswcntrl( )*, *iswdigit( )*, *iswgraph( )*, *iswlower( )*, *iswprint( )*, *iswpunct( )*, *iswspace( )*, *iswupper( )*, *iswxdigit( )*, *setlocale( )*, *uselocale( )*, *wctype( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<locale.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 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 behavior of *n*=0 is now described.

An example is added.

A new function, *iswblank( )*, is added to the list in the APPLICATION USAGE.

Issue 7

The *iswctype_l( )* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

22796 **NAME**

22797 iswdigit, iswdigit_l — test for a decimal digit wide-character code

22798 **SYNOPSIS**

22799
```
#include <wctype.h>
```

22800
```
int iswdigit(wint_t wc);
```

22801 CX
```
int iswdigit_l(wint_t wc, locale_t locale);
```

22802 **DESCRIPTION**

22803 CX For *iswdigit*( ): The functionality described on this reference page is aligned with the ISO C
22804 standard. Any conflict between the requirements described here and the ISO C standard is
22805 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22806 CX The *iswdigit*( ) and *iswdigit_l*( ) functions shall test whether *wc* is a wide-character code
22807 CX representing a character of class **digit** in the current locale of the process, or in the locale
22808 represented by *locale*, respectively; see the Base Definitions volume of IEEE Std 1003.1-200x,
22809 Chapter 7, Locale.

22810 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
22811 code corresponding to a valid character in the current locale, or equal to the value of the macro
22812 WEOF. If the argument has any other value, the behavior is undefined.

22813 **RETURN VALUE**

22814 CX The *iswdigit*( ) and *iswdigit_l*( ) functions shall return non-zero if *wc* is a decimal digit wide-
22815 character code; otherwise, they shall return 0.

22816 **ERRORS**

22817 The *iswdigit_l*( ) function may fail if:

22818 CX [EINVAL] *locale* is not a valid locale object handle.

22819 **EXAMPLES**

22820 None.

22821 **APPLICATION USAGE**

22822 To ensure applications portability, especially across natural languages, only these functions and
22823 the functions in the reference pages listed in the SEE ALSO section should be used for character
22824 classification.

22825 **RATIONALE**

22826 None.

22827 **FUTURE DIRECTIONS**

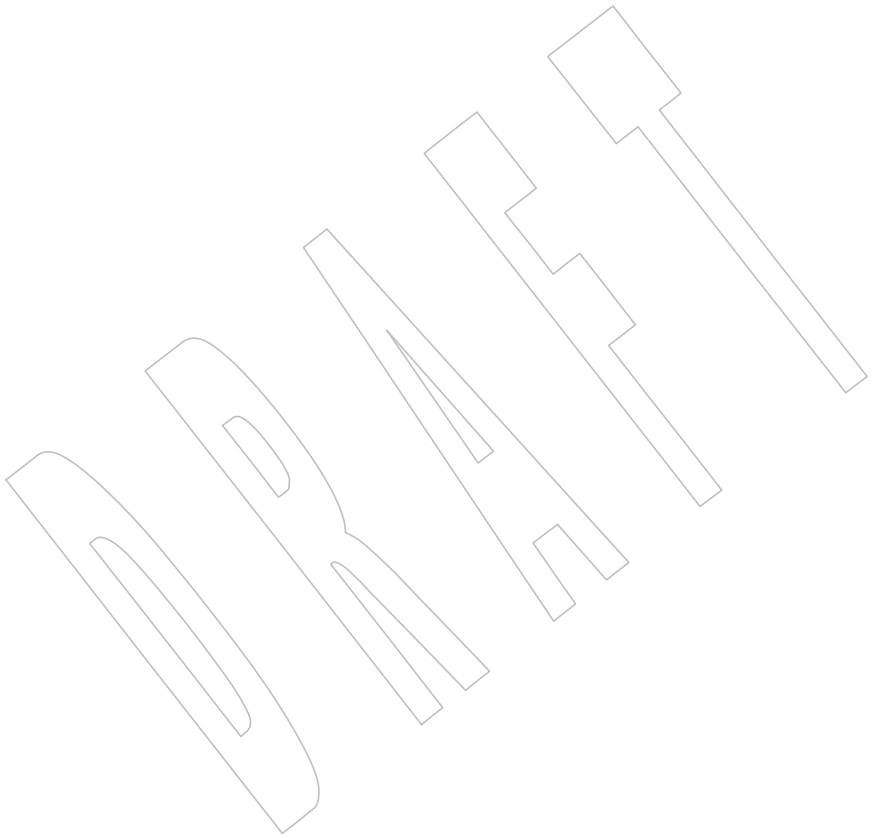
22828 None.

22829 **SEE ALSO**

22830 *iswalnum*( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ),
22831 *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), *uselocale*( ), the Base Definitions volume of
22832 IEEE Std 1003.1-200x, Chapter 7, Locale, **<locale.h>**, **<wctype.h>**

22833 **CHANGE HISTORY**

22834 First released in Issue 4.



22845 **NAME**

22846 iswgraph, iswgraph_l — test for a visible wide-character code

22847 **SYNOPSIS**

22848
```
#include <wctype.h>
```

22849
```
int iswgraph(wint_t wc);
```

22850 CX
```
int iswgraph_l(wint_t wc, locale_t locale);
```

22851 **DESCRIPTION**

22852 CX For *iswgraph*( ): The functionality described on this reference page is aligned with the ISO C
22853 standard. Any conflict between the requirements described here and the ISO C standard is
22854 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22855 CX The *iswgraph*( ) and *iswgraph_l*( ) functions shall test whether *wc* is a wide-character code
22856 CX representing a character of class **graph** in the current locale of the process, or in the locale
22857 represented by *locale*, respectively; see the Base Definitions volume of IEEE Std 1003.1-200x,
22858 Chapter 7, Locale.

22859 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
22860 code corresponding to a valid character in the current locale, or equal to the value of the macro
22861 WEOF. If the argument has any other value, the behavior is undefined.

22862 **RETURN VALUE**

22863 CX The *iswgraph*( ) and *iswgraph_l*( ) functions shall return non-zero if *wc* is a wide-character code
22864 with a visible representation; otherwise, they shall return 0.

22865 **ERRORS**

22866 The *iswgraph_l*( ) function may fail if:

22867 CX [EINVAL] *locale* is not a valid locale object handle.

22868 **EXAMPLES**

22869 None.

22870 **APPLICATION USAGE**

22871 To ensure applications portability, especially across natural languages, only these functions and
22872 the functions in the reference pages listed in the SEE ALSO section should be used for character
22873 classification.

22874 **RATIONALE**

22875 None.

22876 **FUTURE DIRECTIONS**

22877 None.

22878 **SEE ALSO**

22879 *iswalnum*( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ),
22880 *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), *uselocale*( ), the Base Definitions volume of
22881 IEEE Std 1003.1-200x, Chapter 7, Locale, **<locale.h>**, **<wctype.h>**

22882 **CHANGE HISTORY**

22883 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 normative text is updated to avoid use of the term "must" for application requirements.

Issue 7

The *iswgraph_l*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

22894 **NAME**

22895 iswlower, iswlower_l — test for a lowercase letter wide-character code

22896 **SYNOPSIS**

22897
```
#include <wctype.h>
```

22898
```
int iswlower(wint_t wc);
```

22899 CX
```
int iswlower_l(wint_t wc, locale_t locale);
```

22900 **DESCRIPTION**

22901 CX For *iswlower*( ): The functionality described on this reference page is aligned with the ISO C
22902 standard. Any conflict between the requirements described here and the ISO C standard is
22903 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22904 CX The *iswlower*( ) and *iswlower_l*( ) functions shall test whether *wc* is a wide-character code
22905 CX representing a character of class **lower** in the current locale of the process, or in the locale
22906 represented by *locale*, respectively; see the Base Definitions volume of IEEE Std 1003.1-200x,
22907 Chapter 7, Locale.

22908 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
22909 code corresponding to a valid character in the current locale, or equal to the value of the macro
22910 WEOF. If the argument has any other value, the behavior is undefined.

22911 **RETURN VALUE**

22912 CX The *iswlower*( ) and *iswlower_l*( ) functions shall return non-zero if *wc* is a lowercase letter wide-
22913 character code; otherwise, they shall return 0.

22914 **ERRORS**

22915 The *iswlower_l*( ) function may fail if:

22916 CX [EINVAL] *locale* is not a valid locale object handle.

22917 **EXAMPLES**

22918 None.

22919 **APPLICATION USAGE**

22920 To ensure applications portability, especially across natural languages, only these functions and
22921 the functions in the reference pages listed in the SEE ALSO section should be used for character
22922 classification.

22923 **RATIONALE**

22924 None.

22925 **FUTURE DIRECTIONS**

22926 None.

22927 **SEE ALSO**

22928 *iswalnum*( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswprint*( ), *iswpunct*( ),
22929 *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), *uselocale*( ), the Base Definitions volume of
22930 IEEE Std 1003.1-200x, Chapter 7, Locale, **<locale.h>**, **<wctype.h>**

22931 **CHANGE HISTORY**

22932 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):

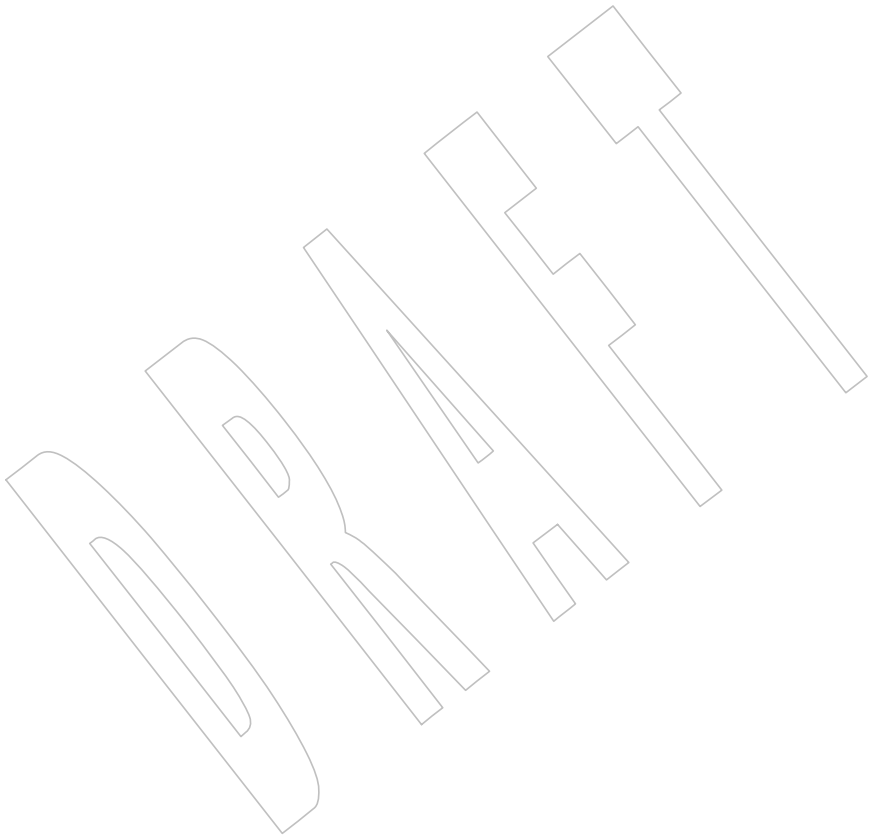
- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

Issue 6

The normative text is updated to avoid use of the term "must" for application requirements.

Issue 7

The *iswlower_l*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.



22943 **NAME**

22944 iswprint, iswprint_l — test for a printable wide-character code

22945 **SYNOPSIS**

22946
```
#include <wctype.h>
```

22947
```
int iswprint(wint_t wc);
```

22948 CX
```
int iswprint_l(wint_t wc, locale_t locale);
```

22949 **DESCRIPTION**

22950 CX For *iswprint*( ): The functionality described on this reference page is aligned with the ISO C
22951 standard. Any conflict between the requirements described here and the ISO C standard is
22952 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

22953 CX The *iswprint*( ) and *iswprint_l*( ) functions shall test whether *wc* is a wide-character code
22954 CX representing a character of class **print** in the current locale of the process, or in the locale
22955 represented by *locale*, respectively; see the Base Definitions volume of IEEE Std 1003.1-200x,
22956 Chapter 7, Locale.

22957 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
22958 code corresponding to a valid character in the current locale, or equal to the value of the macro
22959 WEOF. If the argument has any other value, the behavior is undefined.

22960 **RETURN VALUE**

22961 CX The *iswprint*( ) and *iswprint_l*( ) functions shall return non-zero if *wc* is a printable wide-
22962 character code; otherwise, they shall return 0.

22963 **ERRORS**

22964 The *iswprint_l*( ) function may fail if:

22965 CX [EINVAL] *locale* is not a valid locale object handle.

22966 **EXAMPLES**

22967 None.

22968 **APPLICATION USAGE**

22969 To ensure applications portability, especially across natural languages, only these functions and
22970 the functions in the reference pages listed in the SEE ALSO section should be used for character
22971 classification.

22972 **RATIONALE**

22973 None.

22974 **FUTURE DIRECTIONS**

22975 None.

22976 **SEE ALSO**

22977 *iswalnum*( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswpunct*( ),
22978 *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), *uselocale*( ), the Base Definitions volume of
22979 IEEE Std 1003.1-200x, Chapter 7, Locale, **<locale.h>**, **<wctype.h>**

22980 **CHANGE HISTORY**

22981 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):

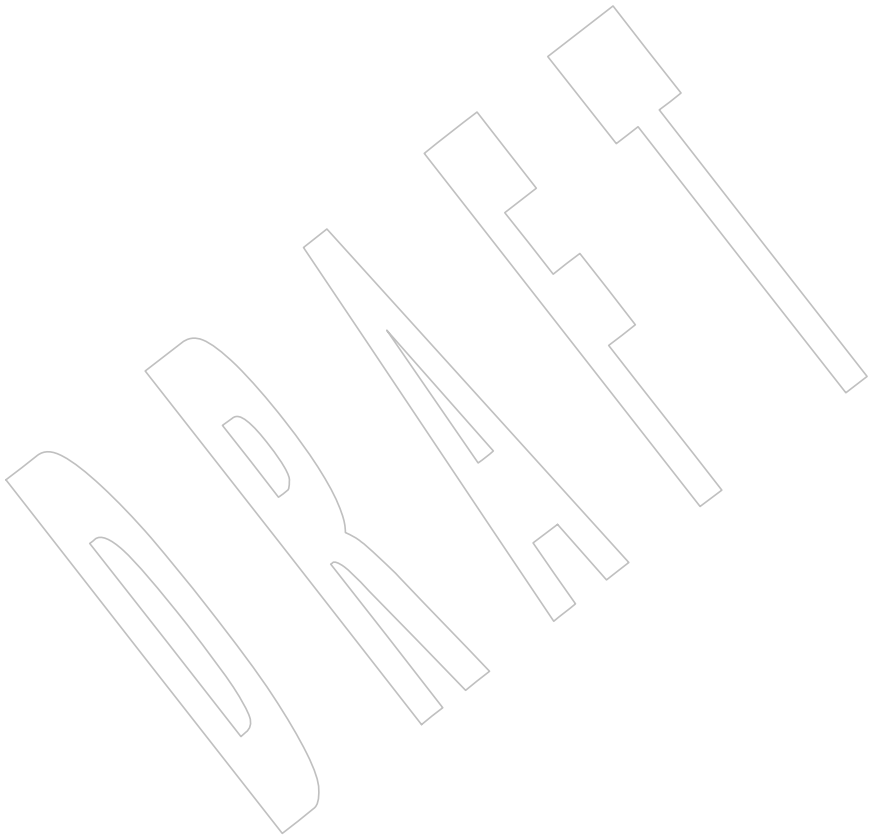
- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

Issue 6

The normative text is updated to avoid use of the term "must" for application requirements.

Issue 7

The *iswprint_l*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.



22992 **NAME**

22993 iswpunct, iswpunct_l — test for a punctuation wide-character code

22994 **SYNOPSIS**

22995
```
#include <wctype.h>
```

22996
```
int iswpunct(wint_t wc);
```

22997 CX
```
int iswpunct_l(wint_t wc, locale_t locale);
```

22998 **DESCRIPTION**

22999 CX For *iswpunct*( ): The functionality described on this reference page is aligned with the ISO C
23000 standard. Any conflict between the requirements described here and the ISO C standard is
23001 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

23002 CX The *iswpunct*( ) and *iswpunct_l*( ) functions shall test whether *wc* is a wide-character code
23003 CX representing a character of class **punct** in the current locale of the process, or in the locale
23004 represented by *locale*, respectively; see the Base Definitions volume of IEEE Std 1003.1-200x,
23005 Chapter 7, Locale.

23006 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
23007 code corresponding to a valid character in the current locale, or equal to the value of the macro
23008 WEOF. If the argument has any other value, the behavior is undefined.

23009 **RETURN VALUE**

23010 CX The *iswpunct*( ) and *iswpunct_l*( ) functions shall return non-zero if *wc* is a punctuation wide-
23011 character code; otherwise, they shall return 0.

23012 **ERRORS**

23013 The *iswpunct_l*( ) function may fail if:

23014 CX [EINVAL] *locale* is not a valid locale object handle.

23015 **EXAMPLES**

23016 None.

23017 **APPLICATION USAGE**

23018 To ensure applications portability, especially across natural languages, only these functions and
23019 the functions in the reference pages listed in the SEE ALSO section should be used for character
23020 classification.

23021 **RATIONALE**

23022 None.

23023 **FUTURE DIRECTIONS**

23024 None.

23025 **SEE ALSO**

23026 *iswalnum*( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ),
23027 *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), *uselocale*( ), the Base Definitions volume of
23028 IEEE Std 1003.1-200x, Chapter 7, Locale, **<locale.h>**, **<wctype.h>**

23029 **CHANGE HISTORY**

23030 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 normative text is updated to avoid use of the term "must" for application requirements.

Issue 7

The *iswpunct_l*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

23041 **NAME**

23042 iswspace, iswspace_l — test for a white-space wide-character code

23043 **SYNOPSIS**

23044
```
#include <wctype.h>
```

23045
```
int iswspace(wint_t wc);
```

23046 CX
```
int iswspace_l(wint_t wc, locale_t locale);
```

23047 **DESCRIPTION**

23048 CX For *iswspace*( ): The functionality described on this reference page is aligned with the ISO C
23049 standard. Any conflict between the requirements described here and the ISO C standard is
23050 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

23051 CX The *iswspace*( ) and *iswspace_l*( ) functions shall test whether *wc* is a wide-character code
23052 CX representing a character of class **space** in the current locale of the process, or in the locale
23053 represented by *locale*, respectively; see the Base Definitions volume of IEEE Std 1003.1-200x,
23054 Chapter 7, Locale.

23055 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
23056 code corresponding to a valid character in the current locale, or equal to the value of the macro
23057 WEOF. If the argument has any other value, the behavior is undefined.

23058 **RETURN VALUE**

23059 CX The *iswspace*( ) and *iswspace_l*( ) functions shall return non-zero if *wc* is a white-space wide-
23060 character code; otherwise, they shall return 0.

23061 **ERRORS**

23062 The *iswspace_l*( ) function may fail if:

23063 CX [EINVAL] *locale* is not a valid locale object handle.

23064 **EXAMPLES**

23065 None.

23066 **APPLICATION USAGE**

23067 To ensure applications portability, especially across natural languages, only these functions and
23068 the functions in the reference pages listed in the SEE ALSO section should be used for character
23069 classification.

23070 **RATIONALE**

23071 None.

23072 **FUTURE DIRECTIONS**

23073 None.

23074 **SEE ALSO**

23075 *iswalnum*( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ),
23076 *iswpunct*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), *uselocale*( ), the Base Definitions volume of
23077 IEEE Std 1003.1-200x, Chapter 7, Locale, **<locale.h>**, **<wctype.h>**

23078 **CHANGE HISTORY**

23079 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):

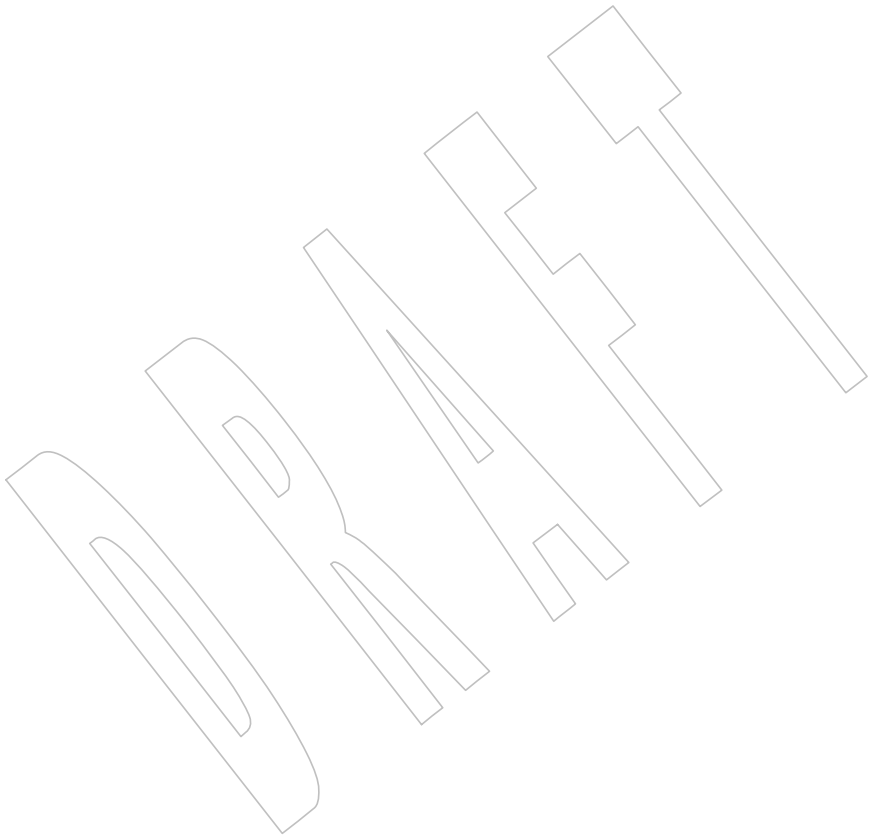
- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

Issue 6

The normative text is updated to avoid use of the term "must" for application requirements.

Issue 7

The *iswspace_l*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.



23090 **NAME**

23091 iswupper, iswupper_l — test for an uppercase letter wide-character code

23092 **SYNOPSIS**

23093
```
#include <wctype.h>
```

23094
```
int iswupper(wint_t wc);
```

23095 CX
```
int iswupper_l(wint_t wc, locale_t locale);
```

23096 **DESCRIPTION**

23097 CX For *iswupper*( ): The functionality described on this reference page is aligned with the ISO C
23098 standard. Any conflict between the requirements described here and the ISO C standard is
23099 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

23100 CX The *iswupper*( ) and *iswupper_l*( ) functions shall test whether *wc* is a wide-character code
23101 CX representing a character of class **upper** in the current locale of the process, or in the locale
23102 represented by *locale*, respectively; see the Base Definitions volume of IEEE Std 1003.1-200x,
23103 Chapter 7, Locale.

23104 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
23105 code corresponding to a valid character in the current locale, or equal to the value of the macro
23106 WEOF. If the argument has any other value, the behavior is undefined.

23107 **RETURN VALUE**

23108 CX The *iswupper*( ) and *iswupper_l*( ) functions shall return non-zero if *wc* is an uppercase letter
23109 wide-character code; otherwise, they shall return 0.

23110 **ERRORS**

23111 The *iswupper_l*( ) function may fail if:

23112 CX [EINVAL] *locale* is not a valid locale object handle.

23113 **EXAMPLES**

23114 None.

23115 **APPLICATION USAGE**

23116 To ensure applications portability, especially across natural languages, only these functions and
23117 the functions in the reference pages listed in the SEE ALSO section should be used for character
23118 classification.

23119 **RATIONALE**

23120 None.

23121 **FUTURE DIRECTIONS**

23122 None.

23123 **SEE ALSO**

23124 *iswalnum*( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ),
23125 *iswpunct*( ), *iswspace*( ), *iswxdigit*( ), *setlocale*( ), *uselocale*( ), the Base Definitions volume of
23126 IEEE Std 1003.1-200x, Chapter 7, Locale, **<locale.h>**, **<wctype.h>**

23127 **CHANGE HISTORY**

23128 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):

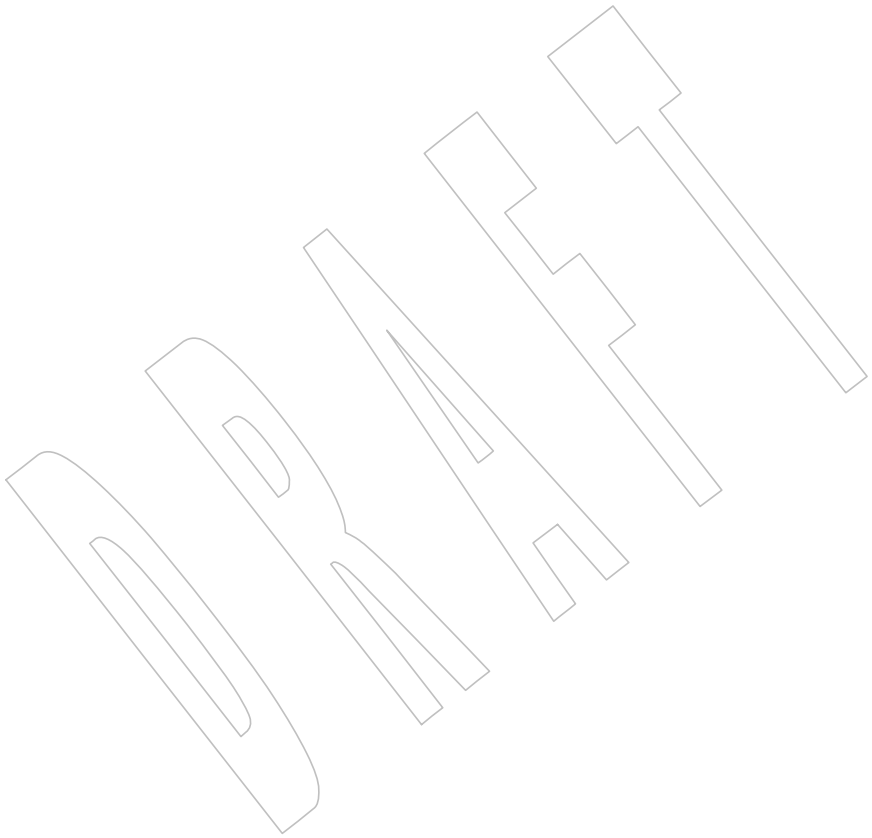
- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

Issue 6

The normative text is updated to avoid use of the term "must" for application requirements.

Issue 7

The *iswupper_l*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.



23139 **NAME**

23140 iswxdigit, iswxdigit_l — test for a hexadecimal digit wide-character code

23141 **SYNOPSIS**

23142
```
#include <wctype.h>
```

23143
```
int iswxdigit(wint_t wc);
```

23144 CX
```
int iswxdigit_l(wint_t wc, locale_t locale);
```

23145 **DESCRIPTION**

23146 CX For *iswxdigit*( ): The functionality described on this reference page is aligned with the ISO C
23147 standard. Any conflict between the requirements described here and the ISO C standard is
23148 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

23149 CX The *iswxdigit*( ) and *iswxdigit_l*( ) functions shall test whether *wc* is a wide-character code
23150 CX representing a character of class **xdigit** in the current locale of the process, or in the locale
23151 represented by *locale*, respectively; see the Base Definitions volume of IEEE Std 1003.1-200x,
23152 Chapter 7, Locale.

23153 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
23154 code corresponding to a valid character in the current locale, or equal to the value of the macro
23155 WEOF. If the argument has any other value, the behavior is undefined.

23156 **RETURN VALUE**

23157 CX The *iswxdigit*( ) and *iswxdigit_l*( ) functions shall return non-zero if *wc* is a hexadecimal digit
23158 wide-character code; otherwise, they shall return 0.

23159 **ERRORS**

23160 The *iswxdigit_l*( ) function may fail if:

23161 CX [EINVAL] *locale* is not a valid locale object handle.

23162 **EXAMPLES**

23163 None.

23164 **APPLICATION USAGE**

23165 To ensure applications portability, especially across natural languages, only these functions and
23166 the functions in the reference pages listed in the SEE ALSO section should be used for character
23167 classification.

23168 **RATIONALE**

23169 None.

23170 **FUTURE DIRECTIONS**

23171 None.

23172 **SEE ALSO**

23173 *iswalnum*( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ),
23174 *iswpunct*( ), *iswspace*( ), *iswupper*( ), *setlocale*( ), *uselocale*( ), the Base Definitions volume of
23175 IEEE Std 1003.1-200x, Chapter 7, Locale, **<locale.h>**, **<wctype.h>**

23176 **CHANGE HISTORY**

23177 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 normative text is updated to avoid use of the term "must" for application requirements.

Issue 7

The *iswxdigit_l*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

NAME

isxdigit, isxdigit_l — test for a hexadecimal digit

SYNOPSIS

```
#include <ctype.h>

int isxdigit(int c);
CX int isxdigit_l(int c, locale_t locale);
```

DESCRIPTION

CX For *isxdigit*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

CX The *isxdigit*( ) and *isxdigit_l*( ) functions shall test whether *c* is a character of class **xdigit** in the CX current locale of the process, or in the locale represented by *locale*, respectively; see the Base Definitions volume of IEEE Std 1003.1-200x, 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

CX The *isxdigit*( ) and *isxdigit_l*( ) functions shall return non-zero if *c* is a hexadecimal digit; otherwise, they shall return 0.

ERRORS

The *isxdigit_l*( ) function may fail if:

CX [EINVAL] *locale* is not a valid locale object handle.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only these functions and the functions in the reference pages listed in the SEE ALSO section should be used for character classification.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalnum( ), *isalpha*( ), *isblank*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale, **<ctype.h>**

CHANGE HISTORY

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

Issue 6

The normative text is updated to avoid use of the term "must" for application requirements.

Issue 7

The *isxdigit_l*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

NAME

j0, j1, jn — Bessel functions of the first kind

SYNOPSIS

XSI

```
#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

feclearexcept( ), *fetestexcept*( ), *isnan*( ), *y0*( ), the Base Definitions volume of IEEE Std 1003.1-200x, 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.

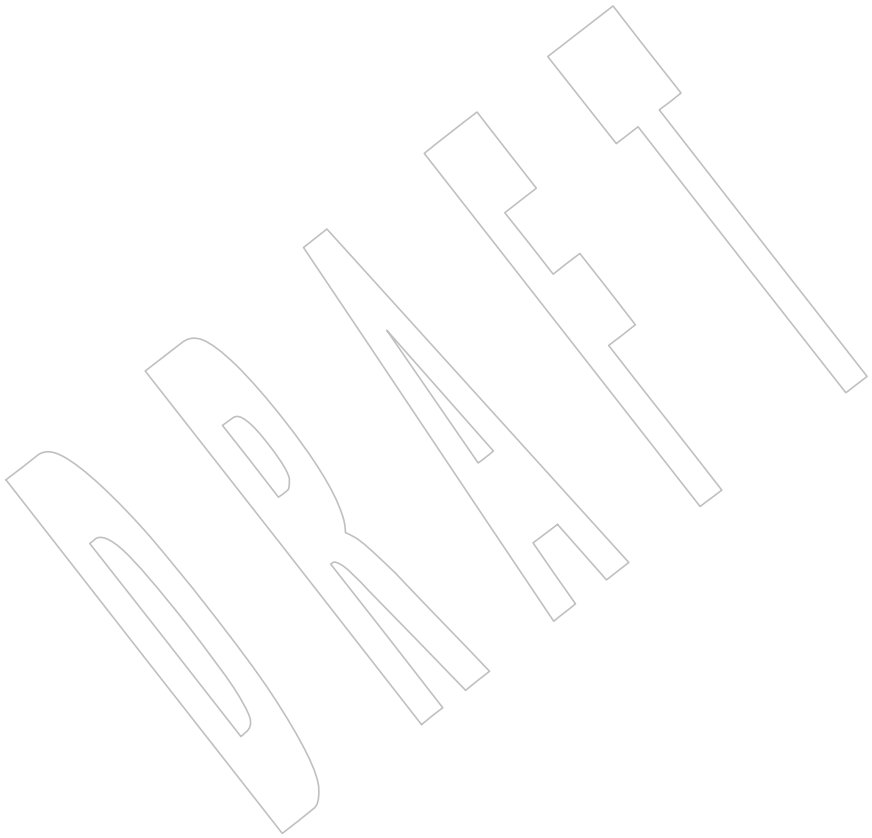
Issue 5

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.



23279 **NAME**

23280 jrand48 — generate a uniformly distributed pseudo-random long signed integer

23281 **SYNOPSIS**

23282 XSI
```
#include <stdlib.h>

long jrand48(unsigned short xsubi[3]);
```

23284 **DESCRIPTION**

23285 Refer to *drand48( )*.

23286 **NAME**

23287 kill — send a signal to a process or a group of processes

23288 **SYNOPSIS**

23289 CX
```
#include <signal.h>

int kill(pid_t pid, int sig);
```

23290

23291 **DESCRIPTION**

23292 The *kill*( ) function shall send a signal to a process or a group of processes specified by *pid*. The
23293 signal to be sent is specified by *sig* and is either one from the list given in **<signal.h>** or 0. If *sig* is
23294 0 (the null signal), error checking is performed but no signal is actually sent. The null signal can
23295 be used to check the validity of *pid*.

23296 For a process to have permission to send a signal to a process designated by *pid*, unless the
23297 sending process has appropriate privileges, the real or effective user ID of the sending process
23298 shall match the real or saved set-user-ID of the receiving process.

23299 If *pid* is greater than 0, *sig* shall be sent to the process whose process ID is equal to *pid*.

23300 If *pid* is 0, *sig* shall be sent to all processes (excluding an unspecified set of system processes)
23301 whose process group ID is equal to the process group ID of the sender, and for which the process
23302 has permission to send a signal.

23303 If *pid* is –1, *sig* shall be sent to all processes (excluding an unspecified set of system processes) for
23304 which the process has permission to send that signal.

23305 If *pid* is negative, but not –1, *sig* shall be sent to all processes (excluding an unspecified set of
23306 system processes) whose process group ID is equal to the absolute value of *pid*, and for which
23307 the process has permission to send a signal.

23308 If the value of *pid* causes *sig* to be generated for the sending process, and if *sig* is not blocked for
23309 the calling thread and if no other thread has *sig* unblocked or is waiting in a *sigwait*( ) function
23310 for *sig*, either *sig* or at least one pending unblocked signal shall be delivered to the sending
23311 thread before *kill*( ) returns.

23312 The user ID tests described above shall not be applied when sending SIGCONT to a process that
23313 is a member of the same session as the sending process.

23314 An implementation that provides extended security controls may impose further
23315 implementation-defined restrictions on the sending of signals, including the null signal. In
23316 particular, the system may deny the existence of some or all of the processes specified by *pid*.

23317 The *kill*( ) function is successful if the process has permission to send *sig* to any of the processes
23318 specified by *pid*. If *kill*( ) fails, no signal shall be sent.

23319 **RETURN VALUE**

23320 Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to
23321 indicate the error.

23322 **ERRORS**

23323 The *kill*( ) function shall fail if:

23324 [EINVAL] The value of the *sig* argument is an invalid or unsupported signal number.

23325 [EPERM] The process does not have permission to send the signal to any receiving
23326 process.

[ESRCH] No process or process group can be found corresponding to that specified by *pid*.

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 1003.1-200x 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 1003.1-200x, although a conforming implementation could provide such an extension.

The unspecified 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 1003.1-200x, 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 1003.1-200x 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 existe6t the exn Tw[(en3A5ty i7e 0 Td[(.l3e40785wo signals t

-I5[(wiAa02 0 Td1X5thatovh n f Td Tw[(er)Tj6a Td(k1)8 .fj128.72 0 tV5

call (subject to permission checking), while others give an error of [ESRCH]. Since the definition of process lifetime in this volume of IEEE Std 1003.1-200x 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 1003.1-200x, **<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 normative text is updated to avoid use of the term "must" for application requirements.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/51 is applied, correcting the RATIONALE section.

NAME

killpg — send a signal to a process group

SYNOPSIS

XSI

```
#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

Sending a Signal to All Other Members of a Process Group

The following example shows how the calling process could send a signal to all other members of its process group. To prevent itself from receiving the signal it first makes itself immune to the signal by ignoring it.

```
#include <signal.h>
#include <unistd.h>
...
    if (signal(SIGUSR1, SIG_IGN) == SIG_ERR)
        /* Handle error */;

    if (killpg(getpgrp(), SIGUSR1) == -1)
        /* Handle error */;"
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getpgid( ), *getpid*( ), *kill*( ), *raise*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/52 is applied, adding the example to the EXAMPLES section.

23450 **NAME**

23451 l64a — convert a 32-bit integer to a radix-64 ASCII string

23452 **SYNOPSIS**

23453 XSI
```
#include <stdlib.h>

char *l64a(long value);
```

23455 **DESCRIPTION**

23456 Refer to *a64l( )*.

23457 **NAME**

23458 labs, llabs — return a long integer absolute value

23459 **SYNOPSIS**

23460
```
#include <stdlib.h>
```

23461
```
long labs(long i);
```
23462
```
long long llabs(long long i);
```

23463 **DESCRIPTION**

23464 CX The functionality described on this reference page is aligned with the ISO C standard. Any
23465 conflict between the requirements described here and the ISO C standard is unintentional. This
23466 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

23467 The *labs*( ) function shall compute the absolute value of the **long** integer operand *i*. The *llabs*( )
23468 function shall compute the absolute value of the **long long** integer operand *i*. If the result
23469 cannot be represented, the behavior is undefined.

23470 **RETURN VALUE**

23471 The *labs*( ) function shall return the absolute value of the **long** integer operand. The *labs*( )
23472 function shall return the absolute value of the **long long** integer operand.

23473 **ERRORS**

23474 No errors are defined.

23475 **EXAMPLES**

23476 None.

23477 **APPLICATION USAGE**

23478 None.

23479 **RATIONALE**

23480 None.

23481 **FUTURE DIRECTIONS**

23482 None.

23483 **SEE ALSO**

23484 *abs*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

23485 **CHANGE HISTORY**

23486 First released in Issue 4. Derived from the ISO C standard.

23487 **Issue 6**

23488 The *llabs*( ) function is added for alignment with the ISO/IEC 9899: 1999 standard.

NAME

lchown — 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.

[EROFS] The file resides on a read-only file system.

The *lchown*( ) function may fail if:

[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.

```
#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 1003.1-200x, **<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.

Issue 7

The *lchown*( ) function is moved from the XSI option to the Base.

23562 **NAME**

23563 lcong48 — seed a uniformly distributed pseudo-random signed long integer generator

23564 **SYNOPSIS**

23565 XSI
```
#include <stdlib.h>

void lcong48(unsigned short param[7]);
```

23567 **DESCRIPTION**

23568 Refer to *drand48( )*.

NAME

ldexp, ldexpf, ldexpl — load exponent of a floating-point number

SYNOPSIS

```
#include <math.h>

double ldexp(double x, int exp);
float ldexpf(float x, int exp);
long double ldexpl(long double x, int exp);
```

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 1003.1-200x defers to the ISO C standard.

These functions shall compute the quantity $x * 2^{exp}$.

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 *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, MX and either 0.0 (if supported), or an implementation-defined value shall be returned.

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

If *x* is ±0 or ±Inf, *x* shall be returned.

If *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_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.

23611 **EXAMPLES**

23612 None.

23613 **APPLICATION USAGE**

23614 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
23615 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

23616 **RATIONALE**

23617 None.

23618 **FUTURE DIRECTIONS**

23619 None.

23620 **SEE ALSO**

23621 *feclearexcept( )*, *fetestexcept( )*, *frexp( )*, *isnan( )*, the Base Definitions volume of
23622 IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
23623 **<math.h>**

23624 **CHANGE HISTORY**

23625 First released in Issue 1. Derived from Issue 1 of the SVID.

23626 **Issue 5**

23627 The DESCRIPTION is updated to indicate how an application should check for an error. This
23628 text was previously published in the APPLICATION USAGE section.

23629 **Issue 6**

23630 The *ldexpf*( ) and *ldexpl*( ) functions are added for alignment with the ISO/IEC 9899: 1999
23631 standard.

23632 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
23633 revised to align with the ISO/IEC 9899: 1999 standard.

23634 IEC 60559: 1989 standard floating-point extensions over the ISO/IEC 9899: 1999 standard are
23635 marked.

23636 **NAME**

23637 ldiv, lldiv — compute quotient and remainder of a long division

23638 **SYNOPSIS**

23639
```
#include <stdlib.h>
```

23640
```
ldiv_t ldiv(long numer, long denom);
```
23641
```
lldiv_t lldiv(long long numer, long long denom);
```

23642 **DESCRIPTION**

23643 CX The functionality described on this reference page is aligned with the ISO C standard. Any
23644 conflict between the requirements described here and the ISO C standard is unintentional. This
23645 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

23646 These functions shall compute the quotient and remainder of the division of the numerator
23647 *numer* by the denominator *denom*. If the division is inexact, the resulting quotient is the **long**
23648 integer (for the *ldiv*( ) function) or **long long** integer (for the *lldiv*( ) function) of lesser magnitude
23649 that is the nearest to the algebraic quotient. If the result cannot be represented, the behavior is
23650 undefined; otherwise, *quot* * *denom*+*rem* shall equal *numer*.

23651 **RETURN VALUE**

23652 The *ldiv*( ) function shall return a structure of type **ldiv_t**, comprising both the quotient and the
23653 remainder. The structure shall include the following members, in any order:

23654
```
long   quot;    /* Quotient */
```
23655
```
long   rem;     /* Remainder */
```

23656 The *lldiv*( ) function shall return a structure of type **lldiv_t**, comprising both the quotient and the
23657 remainder. The structure shall include the following members, in any order:

23658
```
long long   quot;    /* Quotient */
```
23659
```
long long   rem;     /* Remainder */
```

23660 **ERRORS**

23661 No errors are defined.

23662 **EXAMPLES**

23663 None.

23664 **APPLICATION USAGE**

23665 None.

23666 **RATIONALE**

23667 None.

23668 **FUTURE DIRECTIONS**

23669 None.

23670 **SEE ALSO**

23671 *div*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

23672 **CHANGE HISTORY**

23673 First released in Issue 4. Derived from the ISO C standard.

23674 **Issue 6**

23675 The *lldiv*( ) function is added for alignment with the ISO/IEC 9899: 1999 standard.

23676 **NAME**

23677 lfind — find entry in a linear search table

23678 **SYNOPSIS**

23679 XSI
```
#include <search.h>

void *lfind(const void *key, const void *base, size_t *nelp,
    size_t width, int (*compar)(const void *, const void *));
```

23682 **DESCRIPTION**

23683 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);
XSI extern int signgam;
```

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 1003.1-200x defers to the ISO C standard.

These functions shall compute $\log_e|\Gamma(x)|$ where $\Gamma(x)$ is defined as $\int_0^\infty e^{-t}t^{x-1}dt$. The argument x need not be a non-positive integer ($\Gamma(x)$ is defined over the reals, except the non-positive integers).

XSI The sign of $\Gamma(x)$ is returned in the external integer *signgam*.

CX These functions need not be thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

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 logarithmic gamma of x.

If x is a non-positive integer, a pole error shall occur and *lgamma*( ), *lgammaf*( ), and *lgammal*( ) shall return +HUGE_VAL, +HUGE_VALF, and +HUGE_VALL, respectively.

If the correct value would cause overflow, a range error shall occur and *lgamma*( ), *lgammaf*( ), and *lgammal*( ) shall return ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL (having the same sign as the correct value), respectively.

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

If x is 1 or 2, +0 shall be returned.

If x is ±Inf, +Inf shall be returned.

ERRORS

These functions shall fail if:

Pole Error — The x argument is a negative integer or 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 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

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*( ), the Base Definitions volume of IEEE Std 1003.1-200x, 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.

Functionality relating to the XSI option is marked.

NAME

link, linkat — link one file to another file relative to two directory file descriptors

SYNOPSIS

```
#include <unistd.h>

int link(const char *path1, const char *path2);
int linkat(int fd1, const char *path1, int fd2, const char *path2,
    int flag);
```

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.

The *linkat*( ) function shall be equivalent to the *link*( ) function except in the case where either *path1* or *path2* or both are relative paths. In this case a relative path *path1* is interpreted relative to the directory associated with the file descriptor *fd1* instead of the current working directory and similarly for *path2* and the file descriptor *fd2*. It is unspecified whether directory searches are permitted based on whether the file was opened with search permission or on the current permissions of the directory underlying the file descriptor.

Values for *flag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in **<fcntl.h>**:

AT_SYMLINK_FOLLOW
: If *path1* names a symbolic link, a new link for the target of the symbolic link is created.

If *linkat*( ) is passed the special value AT_FDCWD in the *fd1* or *fd2* parameter, the current working directory is used for the respective *path* argument. If both *fd1* and *fd2* have value AT_FDCWD, the behavior shall be identical to a call to *link*( ).

Unless *flag* contains the AT_SYMLINK_FOLLOW flag, if *path1* names a symbolic link, a new link is created for the symbolic link *path1* and not its target.

RETURN VALUE

Upon successful completion, these functions shall return 0. Otherwise, these functions shall return –1 and set *errno* to indicate the error.

ERRORS

These functions 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

23799 by the implementation.

23800 [EEXIST] The *path2* argument resolves to an existing file or refers to a symbolic link.

23801 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path1* or
23802 *path2* argument.

23803 [EMLINK] The number of links to the file named by *path1* would exceed {LINK_MAX}.

23804 [ENAMETOOLONG]
23805 The length of the *path1* or *path2* argument exceeds {PATH_MAX} or a
23806 pathname component is longer than {NAME_MAX}.

23807 [ENOENT] A component of either path prefix does not exist; the file named by *path1* does
23808 not exist; or *path1* or *path2* points to an empty string.

23809 [ENOSPC] The directory to contain the link cannot be extended.

23810 [ENOTDIR] A component of either path prefix is not a directory.

23811 [EPERM] The file named by *path1* is a directory and either the calling process does not
23812 have appropriate privileges or the implementation prohibits using *link*( ) on
23813 directories.

23814 [EROFS] The requested link requires writing in a directory on a read-only file system.

23815 [EXDEV] The link named by *path2* and the file named by *path1* are on different file
23816 systems and the implementation does not support links between file systems.

23817 OB XSR [EXDEV] *path1* refers to a named STREAM.

23818 The *linkat*( ) function shall fail if:

23819 [EBADF] The *path1* or *path2* argument does not specify an absolute path and the *fd1* or
23820 *fd2* argument, respectively, is neither AT_FDCWD nor a valid file descriptor
23821 open for searching.

23822 These functions may fail if:

23823 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
23824 resolution of the *path1* or *path2* argument.

23825 [ENAMETOOLONG]
23826 As a result of encountering a symbolic link in resolution of the *path1* or *path2*
23827 argument, the length of the substituted pathname string exceeded
23828 {PATH_MAX}.

23829 The *linkat*( ) function may fail if:

23830 [EINVAL] The value of the *flag* argument is not valid.

23831 [ENOTDIR] The *path1* or *path2* argument is not an absolute path and *fd1* or *fd2*,
23832 respectively, is neither AT_FDCWD nor a file descriptor associated with a
23833 directory.

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**.

```
#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.

```
#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 1003.1-200x 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.

The purpose of the *linkat*() function is to link files in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to *link*(), resulting in unspecified behavior. By opening a file descriptor for the directory of both the existing file and the target location and using the *linkat*() function it can be guaranteed that the both filenames are in the desired directories.

The AT_SYMLINK_FOLLOW flag allows for implementing both common behaviors of the *link*() function. The POSIX specification requires that if *path1* is a symbolic link, a new link for

the target of the symbolic link is created. Many systems by default or as an alternative provide a mechanism to avoid the implicit symlink lookup and create a new link for the symbolic link itself.

Earlier versions of this standard specified only the *link*( ) function, and required it to behave like *linkat*( ) with the AT_SYMLINK_FOLLOW flag. However, historical practice from SVR4 and Linux kernels had *link*( ) behaving like *linkat*( ) with no flags, and many systems that attempted to provide a conforming *link*( ) function did so in a way that was rarely used, and when it was used did not conform to the standard (e.g., by not being atomic, or by dereferencing the symbolic link incorrectly). Since applications could not rely on *link*( ) following links in practice, the *linkat*( ) function was added taking a flag to specify the desired behavior for the application.

FUTURE DIRECTIONS

None.

SEE ALSO

rename( ), *symlink*( ), *unlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fcntl.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 [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.

Issue 7

SD5-XSH-ERN-93 is applied, adding RATIONALE.

The *linkat*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 2.

Functionality relating to XSI STREAMS is marked obsolescent.

23909 **NAME**

23910 linkat — link one file to another file relative to two directory file descriptors

23911 **SYNOPSIS**

23912
```
#include <unistd.h>
```

23913
```
int linkat(int fd1, const char *path1, int fd2, const char *path2,
    int flag);
```

23915 **DESCRIPTION**

23916 Refer to *link( )*.

23917 **NAME**

23918 lio_listio — list directed I/O

23919 **SYNOPSIS**

23920
```
#include <aio.h>
```

23921
```
int lio_listio(int mode, struct aiocb *restrict const list[restrict],
```
23922
```
    int nent, struct sigevent *restrict sig);
```

23923 **DESCRIPTION**

23924 The *lio_listio*( ) function shall initiate a list of I/O requests with a single function call.

23925 The *mode* argument takes one of the values LIO_WAIT or LIO_NOWAIT declared in **<aio.h>** and
23926 determines whether the function returns when the I/O operations have been completed, or as
23927 soon as the operations have been queued. If the *mode* argument is LIO_WAIT, the function shall
23928 wait until all I/O is complete and the *sig* argument shall be ignored.

23929 If the *mode* argument is LIO_NOWAIT, the function shall return immediately, and asynchronous
23930 notification shall occur, according to the *sig* argument, when all the I/O operations complete. If
23931 *sig* is NULL, then no asynchronous notification shall occur. If *sig* is not NULL, asynchronous
23932 notification occurs as specified in Section 2.4.1 when all the requests in *list* have completed.

23933 The I/O requests enumerated by *list* are submitted in an unspecified order.

23934 The *list* argument is an array of pointers to **aiocb** structures. The array contains *nent* elements.
23935 The array may contain NULL elements, which shall be ignored.

23936 If the buffer pointed to by *list* or the **aiocb** structures pointed to by the elements of the array *list*
23937 become illegal addresses before all asynchronous I/O completed and, if necessary, the
23938 notification is sent, then the behavior is undefined. If the buffers pointed to by the *aio_buf*
23939 member of the **aiocb** structure pointed to by the elements of the array *list* become illegal
23940 addresses prior to the asynchronous I/O associated with that **aiocb** structure being completed,
23941 the behavior is undefined.

23942 The *aio_lio_opcode* field of each **aiocb** structure specifies the operation to be performed. The
23943 supported operations are LIO_READ, LIO_WRITE, and LIO_NOP; these symbols are defined in
23944 **<aio.h>**. The LIO_NOP operation causes the list entry to be ignored. If the *aio_lio_opcode*
23945 element is equal to LIO_READ, then an I/O operation is submitted as if by a call to *aio_read*( )
23946 with the *aiocbp* equal to the address of the **aiocb** structure. If the *aio_lio_opcode* element is equal to
23947 LIO_WRITE, then an I/O operation is submitted as if by a call to *aio_write*( ) with the *aiocbp*
23948 equal to the address of the **aiocb** structure.

23949 The *aio_fildes* member specifies the file descriptor on which the operation is to be performed.

23950 The *aio_buf* member specifies the address of the buffer to or from which the data is transferred.

23951 The *aio_nbytes* member specifies the number of bytes of data to be transferred.

23952 The members of the **aiocb** structure further describe the I/O operation to be performed, in a
23953 manner identical to that of the corresponding **aiocb** structure when used by the *aio_read*( ) and
23954 *aio_write*( ) functions.

23955 The *nent* argument specifies how many elements are members of the list; that is, the length of the
23956 array.

23957 The behavior of this function is altered according to the definitions of synchronized I/O data
23958 integrity completion and synchronized I/O file integrity completion if synchronized I/O is
23959 enabled on the file associated with *aio_fildes*.

23960 For regular files, no data transfer shall occur past the offset maximum established in the open

file description associated with *aiocbp->aio_fildes*.

If *sig->sigev_notify* is SIGEV_THREAD and *sig->sigev_notify_attributes* is a non-NULL pointer and the block pointed to by this pointer becomes an illegal address prior to all asynchronous I/O being completed, then the behavior is undefined.

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_lio_opcode* is LIO_READ, the file is a regular file, *aiocbp->aio_nbytes* 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_lio_opcode* field is invalid, and *aio_error*( ) returns [EINVAL]) or might arise from application behavior (for example, the file descriptor is closed before the operation is initiated, and *aio_error*( ) returns [EBADF]).

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 1003.1-200x, **<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 normative text 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.

Issue 6

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/53 is applied, adding new text for symmetry with the *aio_read*( ) and *aio_write*( ) functions to the DESCRIPTION.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/54 is applied, adding text to the DESCRIPTION making it explicit that the user is required to keep the structure pointed to by *sig->sigev_notify_attributes* valid until the last asynchronous operation finished and the notification has been sent.

Issue 7

The *lio_listio*( ) function is moved from the Asynchronous Input and Output option to the Base.

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 1003.1-200x, **<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.

24115 **NAME**

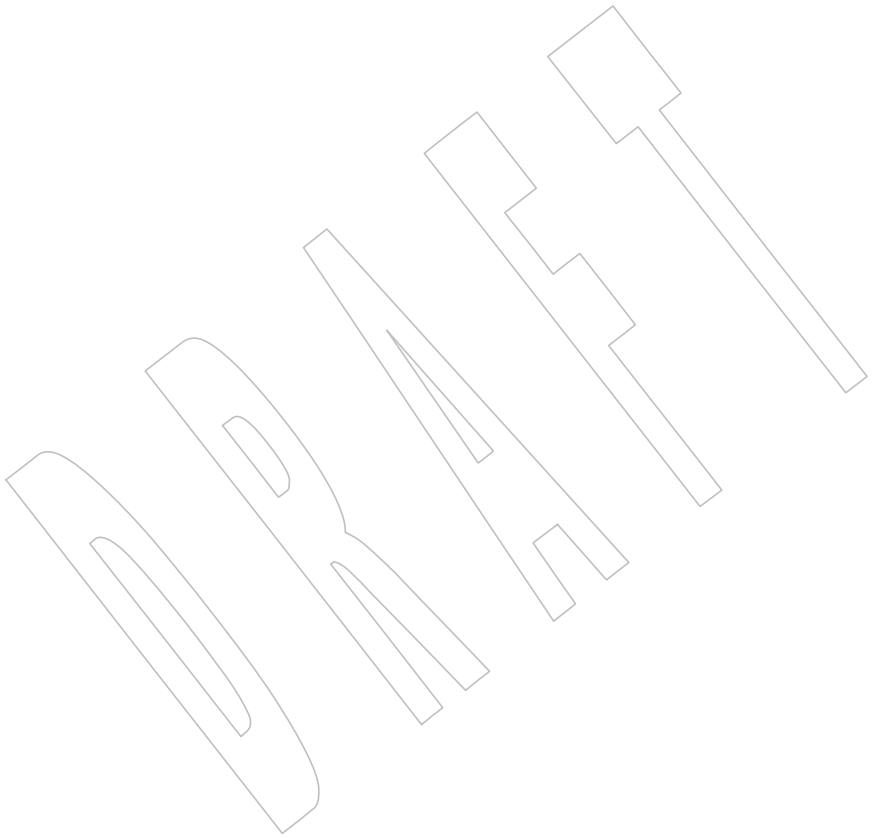
24116 llabs — return a long integer absolute value

24117 **SYNOPSIS**

24118 ```
#include <stdlib.h>
```

24119 ```
long long llabs(long long i);
```

24120 **DESCRIPTION**

24121 Refer to *labs( )*.



24122 **NAME**

24123 lldiv — compute quotient and remainder of a long division

24124 **SYNOPSIS**

24125
```
#include <stdlib.h>
```

24126
```
lldiv_t lldiv(long long numer, long long denom);
```

24127 **DESCRIPTION**

24128 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

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 1003.1-200x 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.

MX 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**, an unspecified value
MX shall be returned. On systems that support the IEC 60559 Floating-Point option, a domain error shall occur;
CX otherwise, a domain error may occur.

If the correct value is negative and too large to represent as a **long long**, an unspecified value
MX shall be returned. On systems that support the IEC 60559 Floating-Point option, a domain error shall occur;
CX otherwise, a domain error may occur.

ERRORS

These functions shall fail if:

MX 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.

These functions may fail if:

Domain Error — 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 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

Issue 7

ISO C TC2 #53 is applied.

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

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 1003.1-200x 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.

MX 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**, an unspecified value
MX shall be returned. On systems that support the IEC 60559 Floating-Point option, a domain error shall occur;
CX otherwise, a domain error may occur.

If the correct value is negative and too large to represent as a **long long**, an unspecified value
MX shall be returned. On systems that support the IEC 60559 Floating-Point option, a domain error shall occur;
CX otherwise, a domain error may occur.

ERRORS

These functions shall fail if:

MX 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.

These functions may fail if:

Domain Error — 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 *llrint*( ) 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 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

Issue 7

ISO/IEC 9899: 1999 standard, Technical Corrigendum 2 #54 (SD5-XSH-ERN-75) is applied.

NAME

localeconv — return locale-specific information

SYNOPSIS

```
#include <locale.h>

struct lconv *localeconv(void);
```

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 1003.1-200x 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: 2001 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** 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 a value indicating the separation of the **currency_symbol**, the sign string, and the value for a non-negative formatted monetary quantity.

char n_cs_precedes
Set to 1 if the **currency_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 a value indicating the separation of the **currency_symbol**, the sign string, and the value for a negative formatted monetary quantity.

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 formatted monetary quantity.

The elements of **grouping** and **mon_grouping** are interpreted according to the following:

24343 {CHAR_MAX} No further grouping is to be performed.

24344 0 The previous element is to be repeatedly used for the remainder of the digits.

24345 24346 24347 *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.

24348 24349 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:

24350 0 No space separates the currency symbol and value.

24351 24352 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.

24353 24354 2 If the currency symbol and sign string are adjacent, a space separates them; otherwise, a space separates the sign string from the value.

24355 24356 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.

24357 24358 The values of **p_sign_posn**, **n_sign_posn**, **int_p_sign_posn**, and **int_n_sign_posn** are interpreted according to the following:

24359 0 Parentheses surround the quantity and **currency_symbol** or **int_curr_symbol**.

24360 1 The sign string precedes the quantity and **currency_symbol** or **int_curr_symbol**.

24361 2 The sign string succeeds the quantity and **currency_symbol** or **int_curr_symbol**.

24362 3 The sign string immediately precedes the **currency_symbol** or **int_curr_symbol**.

24363 4 The sign string immediately succeeds the **currency_symbol** or **int_curr_symbol**.

24364 24365 The implementation shall behave as if no function in this volume of IEEE Std 1003.1-200x calls *localeconv*( ).

24366 24367 CX The *localeconv*( ) function need not be thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

24368 **RETURN VALUE**

24369 24370 24371 24372 24373 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* or calls to *uselocale*( ) which change the categories *LC_MONETARY* or *LC_NUMERIC* may overwrite the contents of the structure.

24374 **ERRORS**

24375 No errors are defined.

24376 **EXAMPLES**

24377 None.

24378 **APPLICATION USAGE**

24379 24380 The following table illustrates the rules which may be used by four countries to format monetary quantities.

24381

Country	Positive Format	Negative Format	International Format
Italy	L.1.230	–L.1.230	ITL.1.230
Netherlands	F 1.234,56	F –1.234,56	NLG 1.234,56
Norway	kr1.234,56	kr1.234,56–	NOK 1.234,56
Switzerland	SFrs.1,234.56	SFrs.1,234.56C	CHF 1,234.56

For these four countries, the respective values for the monetary members of the structure returned by *localeconv*( ) are:

	Italy	Netherlands	Norway	Switzerland
int_curr_symbol	"ITL."	"NLG "	"NOK "	"CHF "
currency_symbol	"L."	"F"	"kr"	"SFrs."
mon_decimal_point	""	","	","	"."
mon_thousands_sep	"."	"."	"."	","
mon_grouping	"\3"	"\3"	"\3"	"\3"
positive_sign	""	""	""	""
negative_sign	"-"	"-"	"-"	"C"
int_frac_digits	0	2	2	2
frac_digits	0	2	2	2
p_cs_precedes	1	1	1	1
p_sep_by_space	0	1	0	0
n_cs_precedes	1	1	1	1
n_sep_by_space	0	1	0	0
p_sign_posn	1	1	1	1
n_sign_posn	1	4	2	2
int_p_cs_precedes	1	1	1	1
int_n_cs_precedes	1	1	1	1
int_p_sep_by_space	0	0	0	0
int_n_sep_by_space	0	0	0	0
int_p_sign_posn	1	1	1	1
int_n_sign_posn	1	4	4	2

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalpha( ), *isascii*( ), *nl_langinfo*( ), *printf*( ), *scanf*( ), *setlocale*( ), *strcat*( ), *strchr*( ), *strcmp*( ), *strcoll*( ), *strcpy*( ), *strftime*( ), *strlen*( ), *strpbrk*( ), *strspn*( ), *strtok*( ), *strxfrm*( ), *strtod*( ), *uselocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<langinfo.h>**, **<locale.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the ANSI C standard.

Issue 6

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.

ISO/IEC 9899: 1999 standard, Technical Corrigendum 1 is incorporated.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/31 is applied, removing references to **int_curr_symbol** and updating the descriptions of **p_sep_by_space** and **n_sep_by_space**. These changes are for alignment with the ISO C 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);
CX struct tm *localtime_r(const time_t *restrict timer,
       struct tm *restrict result);
```

DESCRIPTION

CX 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 1003.1-200x 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
CX 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 1003.1-200x, 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 thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

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*.

If the reentrant version does not set *tzname*, it shall not set *daylight* and shall not set *timezone*.

RETURN VALUE

Upon successful completion, the *localtime*( ) function shall return a pointer to the broken-down
CX time structure. If an error is detected, *localtime*( ) shall return a null pointer and set *errno* to indicate the error.

Upon successful completion, *localtime_r*( ) shall return a pointer to the structure pointed to by the argument *result*. If an error is detected, *localtime_r*( ) s

ERRORS

CX The *localtime*( ) and *localtime_r*( ) functions shall fail if:

CX [EOVERFLOW] The result cannot be represented.

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.

```
#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:

```
Wed Jun 26 10:32:15 1996
835810335 secs since the Epoch
```

Getting the Modification Time for a File

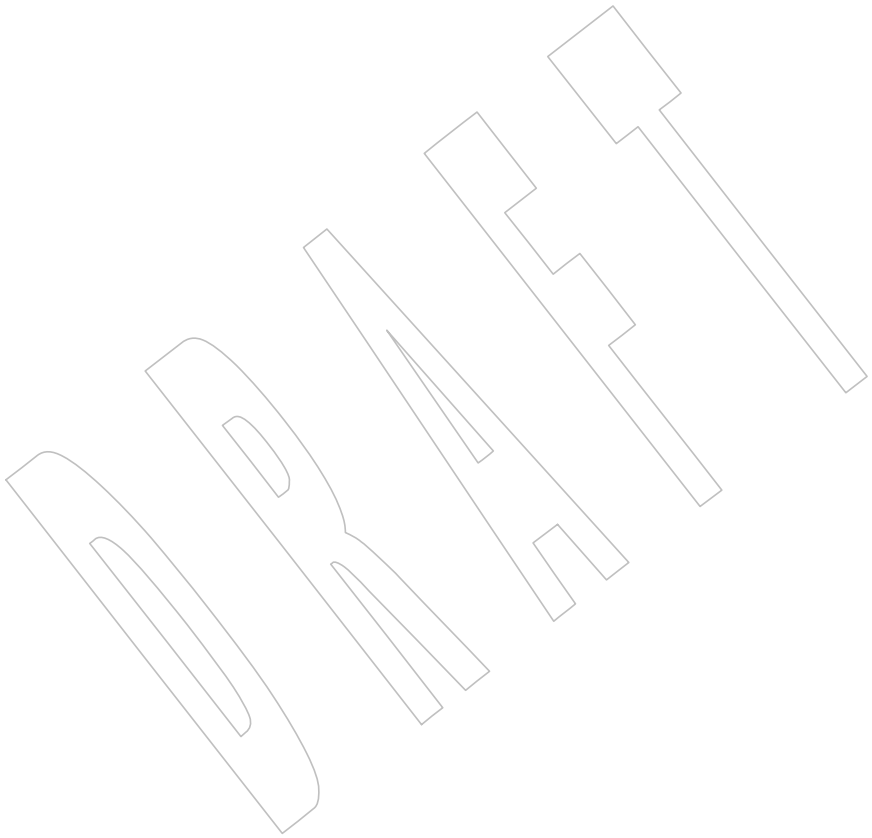
The following example gets the modification time for a file. The *localtime*( ) function converts the **time_t** value of the last modification date, obtained by a previous call to *stat*( ), into a **tm** structure that contains the year, month, day, and so on.

```
#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.

```
#include <time.h>
#include <stdio.h>
...
time_t now;
int minutes_to_event;
...
time(&now);
printf("The time is ");
```

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 threads in other processes which attempt to lock the locked file section shall either return an error value or block until the section becomes unlocked. All the locks for a process are removed when the process terminates. Record locking with *lockf*( ) shall be supported for regular files and may be supported for other files.

The *fildes* argument is an open file descriptor. To establish a lock with this function, the file descriptor shall be opened with write-only permission (O_WRONLY) or with read/write permission (O_RDWR).

The *function* argument is a control value which specifies the action to be taken. The permissible values for *function* are defined in **<unistd.h>** as follows:

Function	Description
F_ULOCK	Unlock locked sections.
F_LOCK	Lock a section for exclusive use.
F_TLOCK	Test and lock a section for exclusive use.
F_TEST	Test a section for locks by other processes.

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 a locked section of another process. 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.

[EINTR] A signal was caught during execution of the function.

[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 1003.1-200x, **<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 normative text is updated to avoid use of the term "must" for application requirements.

Issue 7

Austin Group Interpretation 1003.1-2001 #054 is applied, updating the DESCRIPTION.

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

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 1003.1-200x defers to the ISO C standard.

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 *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 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.

MX 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.

MX 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:

MX Domain Error — The finite value of x is negative, 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.

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 1003.1-200x, 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 normative text 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.

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

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

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 1003.1-200x 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 *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 base 10 logarithm of x.

If x is ±0, a pole error shall occur and *log10*( ), *log10f*( ), and *log10l*( ) shall return –HUGE_VAL, –HUGE_VALF, and –HUGE_VALL, respectively.

MX 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.

MX 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.

ERRORS

These functions shall fail if:

MX Domain Error — The finite value of x is negative, 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.

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 1003.1-200x, 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 normative text 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

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 1003.1-200x defers to the ISO C standard.

These functions shall compute $\log_e(1.0 + x)$.

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 natural logarithm of $1.0 + x$.

If x is –1, a pole error shall occur and *log1p*( ), *log1pf*( ), and *log1pl*( ) shall return –HUGE_VAL, –HUGE_VALF, and –HUGE_VALL, respectively.

MX For finite values of x that are less than –1, or if x is –Inf, 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 ±0, or +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:

MX Domain Error — The finite value of x is less than –1, 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 –1.

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.

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 (*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

feclearexcept( ), *fetestexcept*( ), *log*( ), the Base Definitions volume of IEEE Std 1003.1-200x, 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 normative text 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.

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

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

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 1003.1-200x 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 *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 base 2 logarithm of x.

If x is ±0, a pole error shall occur and *log2*( ), *log2f*( ), and *log2l*( ) shall return –HUGE_VAL, –HUGE_VALF, and –HUGE_VALL, respectively.

MX 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.

MX 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:

MX Domain Error — The finite value of x is less than zero, 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.

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 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

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

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 1003.1-200x 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:

$$1 <= x * FLT\_RADIX^{-logb(x)} < FLT\_RADIX$$

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 exponent of x.

If x is ±0, *logb*(), *logbf*(), and *logbl*() shall return –HUGE_VAL, –HUGE_VALF, and MX –HUGE_VALL, respectively. On systems that support the IEC 60559 Floating-Point option, a pole error shall occur;
CX otherwise, a pole error may occur.

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

MX If x is ±Inf, +Inf shall be returned.

ERRORS

These functions shall fail if:

MX Pole Error — The value of x is ±0.

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.

These functions may fail if:

Pole Error — The value of x is 0.

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

feclearexcept( ), *fetestexcept( )*, *ilogb( )*, *scalbln( )*, the Base Definitions volume of IEEE Std 1003.1-200x, 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.

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

Issue 7

ISO/IEC 9899: 1999 standard, Technical Corrigendum 2 #50 (SD5-XSH-ERN-76) is applied.

24983 **NAME**

24984 logf, logl — natural logarithm function

24985 **SYNOPSIS**

24986
```
#include <math.h>
```

24987
```
float logf(float x);
```

24988
```
long double logl(long double x);
```

24989 **DESCRIPTION**

24990 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 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

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 CX 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).

25033 **RATIONALE**

25034 None.

25035 **FUTURE DIRECTIONS**

25036 None.

25037 **SEE ALSO**

25038 *setjmp( )*, *sigaction( )*, *siglongjmp( )*, *sigsetjmp( )*, the Base Definitions volume of
25039 IEEE Std 1003.1-200x, **<setjmp.h>**

25040 **CHANGE HISTORY**

25041 First released in Issue 1. Derived from Issue 1 of the SVID.

25042 **Issue 5**

25043 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

25044 **Issue 6**

25045 Extensions beyond the ISO C standard are marked.

25046 The following new requirements on POSIX implementations derive from alignment with the
25047 Single UNIX Specification:

25048 • The DESCRIPTION now explicitly makes *longjmp*( )'s effect on the signal mask
25049 unspecified.

25050 The DESCRIPTION is updated for alignment with the ISO/IEC 9899: 1999 standard.

25051 **NAME**

25052 lrand48 — generate uniformly distributed pseudo-random non-negative long integers

25053 **SYNOPSIS**

25054 XSI
```
#include <stdlib.h>

long lrand48(void);
```

25055

25056 **DESCRIPTION**

25057 Refer to *drand48( )*.

25058 **NAME**

25059 lrint, lrintf, lrintl — round to nearest integer value using current rounding direction

25060 **SYNOPSIS**

25061
```
#include <math.h>

long lrint(double x);
long lrintf(float x);
long lrintl(long double x);
```

25065 **DESCRIPTION**

25066 CX The functionality described on this reference page is aligned with the ISO C standard. Any
25067 conflict between the requirements described here and the ISO C standard is unintentional. This
25068 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

25069 These functions shall round their argument to the nearest integer value, rounding according to
25070 the current rounding direction.

25071 An application wishing to check for error situations should set *errno* to zero and call
25072 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
25073 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
25074 zero, an error has occurred.

25075 **RETURN VALUE**

25076 Upon successful completion, these functions shall return the rounded integer value.

25077 MX If *x* is NaN, a domain error shall occur and an unspecified value is returned.

25078 If *x* is +Inf, a domain error shall occur and an unspecified value is returned.

25079 If *x* is –Inf, a domain error shall occur and an unspecified value is returned.

25080 If the correct value is positive and too large to represent as a **long**, an unspecified value shall be
25081 MX returned. On systems that support the IEC 60559 Floating-Point option, a domain error shall
25082 occur;
25083 CX otherwise, a domain error may occur.

25084 If the correct value is negative and too large to represent as a **long**, an unspecified value shall be
25085 MX returned. On systems that support the IEC 60559 Floating-Point option, a domain error shall
25086 occur;
25087 CX otherwise, a domain error may occur.

25088 **ERRORS**

25089 These functions shall fail if:

25090 MX Domain Error The *x* argument is NaN or ±Inf, or the correct value is not representable as an
25091 integer.

25092 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
25093 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
25094 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
25095 shall be raised.

25096 These functions may fail if:

25097 Domain Error The correct value is not representable as an integer.

25098 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
25099 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
25100 & 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 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

Issue 7

ISO/IEC 9899: 1999 standard, Technical Corrigendum 2 #53 (SD5-XSH-ERN-77) is applied.

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

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 1003.1-200x 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.

MX 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**, an unspecified value shall be returned. MX On systems that support the IEC 60559 Floating-Point option, a domain shall occur; CX otherwise, a domain error may occur.

If the correct value is negative and too large to represent as a **long**, an unspecified value shall be returned. MX On systems that support the IEC 60559 Floating-Point option, a domain shall occur; CX otherwise, a domain error may occur.

ERRORS

These functions shall fail if:

MX 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.

These functions may fail if:

Domain Error — 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.

25164 **EXAMPLES**

25165 None.

25166 **APPLICATION USAGE**

25167 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
25168 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

25169 **RATIONALE**

25170 These functions differ from the *lrint*( ) functions in the default rounding direction, with the
25171 *lround*( ) functions rounding halfway cases away from zero and needing not to raise the inexact
25172 floating-point exception for non-integer arguments that round to within the range of the return
25173 type.

25174 **FUTURE DIRECTIONS**

25175 None.

25176 **SEE ALSO**

25177 *feclearexcept*( ), *fetestexcept*( ), *llround*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
25178 Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

25179 **CHANGE HISTORY**

25180 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

25181 **Issue 7**

25182 ISO/IEC 9899: 1999 standard, Technical Corrigendum 2 #54 (SD5-XSH-ERN-78) is applied.

NAME

lsearch, lfind — linear search and update

SYNOPSIS

XSI

```
#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.

```
#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 1003.1-200x, **<search.h>**

CHANGE HISTORY

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

Issue 6

The normative text 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 1003.1-200x 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 1003.1-200x, **<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.

25326 **NAME**

25327 lstat — get file status

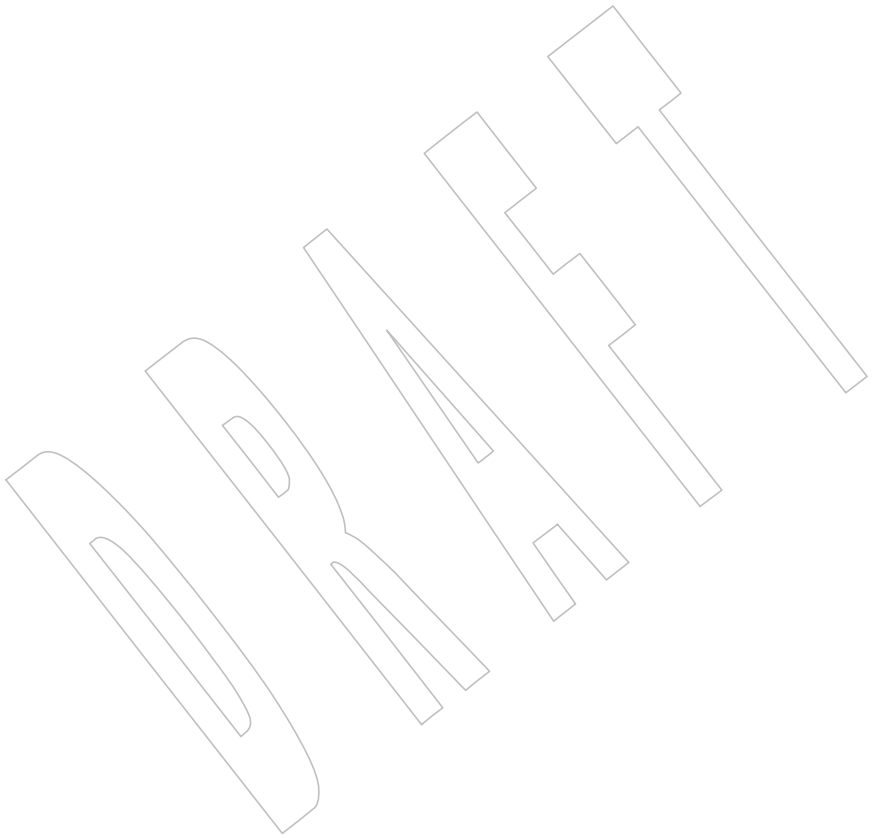
25328 **SYNOPSIS**

25329 `#include <sys/stat.h>`

25330 `int lstat(const char *restrict path, struct stat *restrict buf);`

25331 **DESCRIPTION**

25332 Refer to *fstatat( )*.



NAME

malloc — a memory allocator

SYNOPSIS

```
#include <stdlib.h>

void *malloc(size_t size);
```

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 1003.1-200x 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

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.

25379 **NAME**

25380 mblen — get number of bytes in a character

25381 **SYNOPSIS**

25382
```
#include <stdlib.h>
```

25383
```
int mblen(const char *s, size_t n);
```

25384 **DESCRIPTION**

25385 CX The functionality described on this reference page is aligned with the ISO C standard. Any
25386 conflict between the requirements described here and the ISO C standard is unintentional. This
25387 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

25388 If *s* is not a null pointer, *mblen*( ) shall determine the number of bytes constituting the character
25389 pointed to by *s*. Except that the shift state of *mbtowc*( ) is not affected, it shall be equivalent to:

25390
```
mbtowc((wchar_t *)0, s, n);
```

25391 The implementation shall behave as if no function defined in this volume of
25392 IEEE Std 1003.1-200x calls *mblen*( ).

25393 The behavior of this function is affected by the *LC_CTYPE* category of the current locale. For a
25394 state-dependent encoding, this function shall be placed into its initial state by a call for which its
25395 character pointer argument, *s*, is a null pointer. Subsequent calls with *s* as other than a null
25396 pointer shall cause the internal state of the function to be altered as necessary. A call with *s* as a
25397 null pointer shall cause this function to return a non-zero value if encodings have state
25398 dependency, and 0 otherwise. If the implementation employs special bytes to change the shift
25399 state, these bytes shall not produce separate wide-character codes, but shall be grouped with an
25400 adjacent character. Changing the *LC_CTYPE* category causes the shift state of this function to be
25401 unspecified.

25402 **RETURN VALUE**

25403 If *s* is a null pointer, *mblen*( ) shall return a non-zero or 0 value, if character encodings,
25404 respectively, do or do not have state-dependent encodings. If *s* is not a null pointer, *mblen*( ) shall
25405 either return 0 (if *s* points to the null byte), or return the number of bytes that constitute the
25406 character (if the next *n* or fewer bytes form a valid character), or return –1 (if they do not form a
25407 CX valid character) and may set *errno* to indicate the error. In no case shall the value returned be
25408 greater than *n* or the value of the {MB_CUR_MAX} macro.

25409 **ERRORS**

25410 The *mblen*( ) function may fail if:

25411 XSI [EILSEQ] Invalid character sequence is detected.

25412 **EXAMPLES**

25413 None.

25414 **APPLICATION USAGE**

25415 None.

25416 **RATIONALE**

25417 None.

25418 **FUTURE DIRECTIONS**

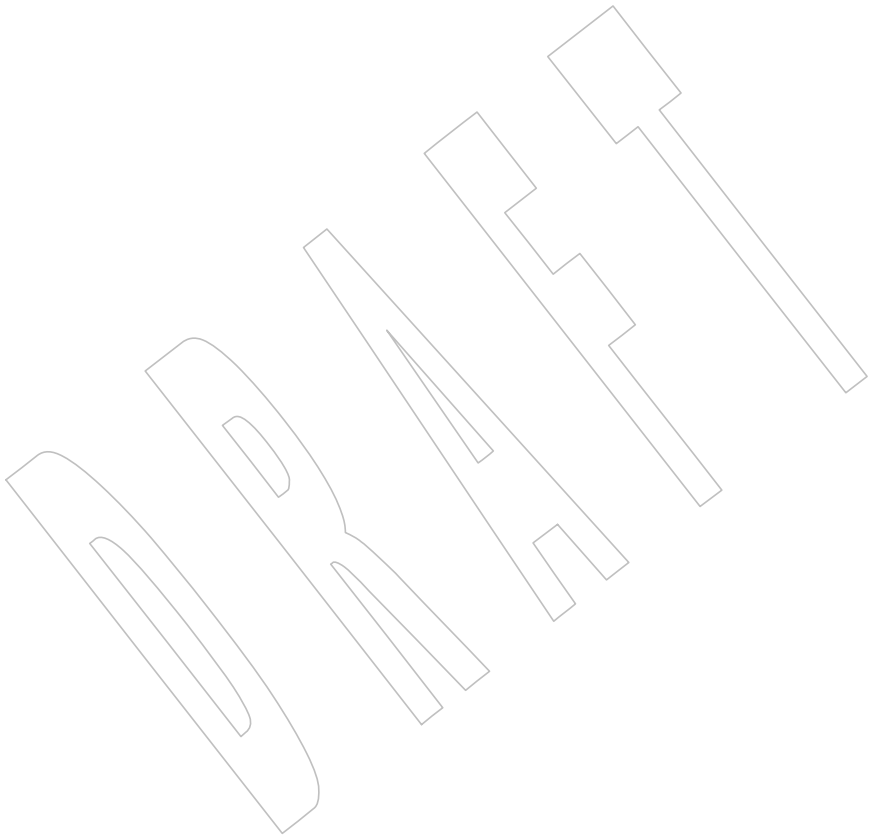
25419 None.

25420 **SEE ALSO**

25421 *mbtowc( )*, *mbstowcs( )*, *wctomb( )*, *wcstombs( )*, the Base Definitions volume of
25422 IEEE Std 1003.1-200x, **<stdlib.h>**

25423 **CHANGE HISTORY**

25424 First released in Issue 4. Aligned with the ISO C standard.



NAME

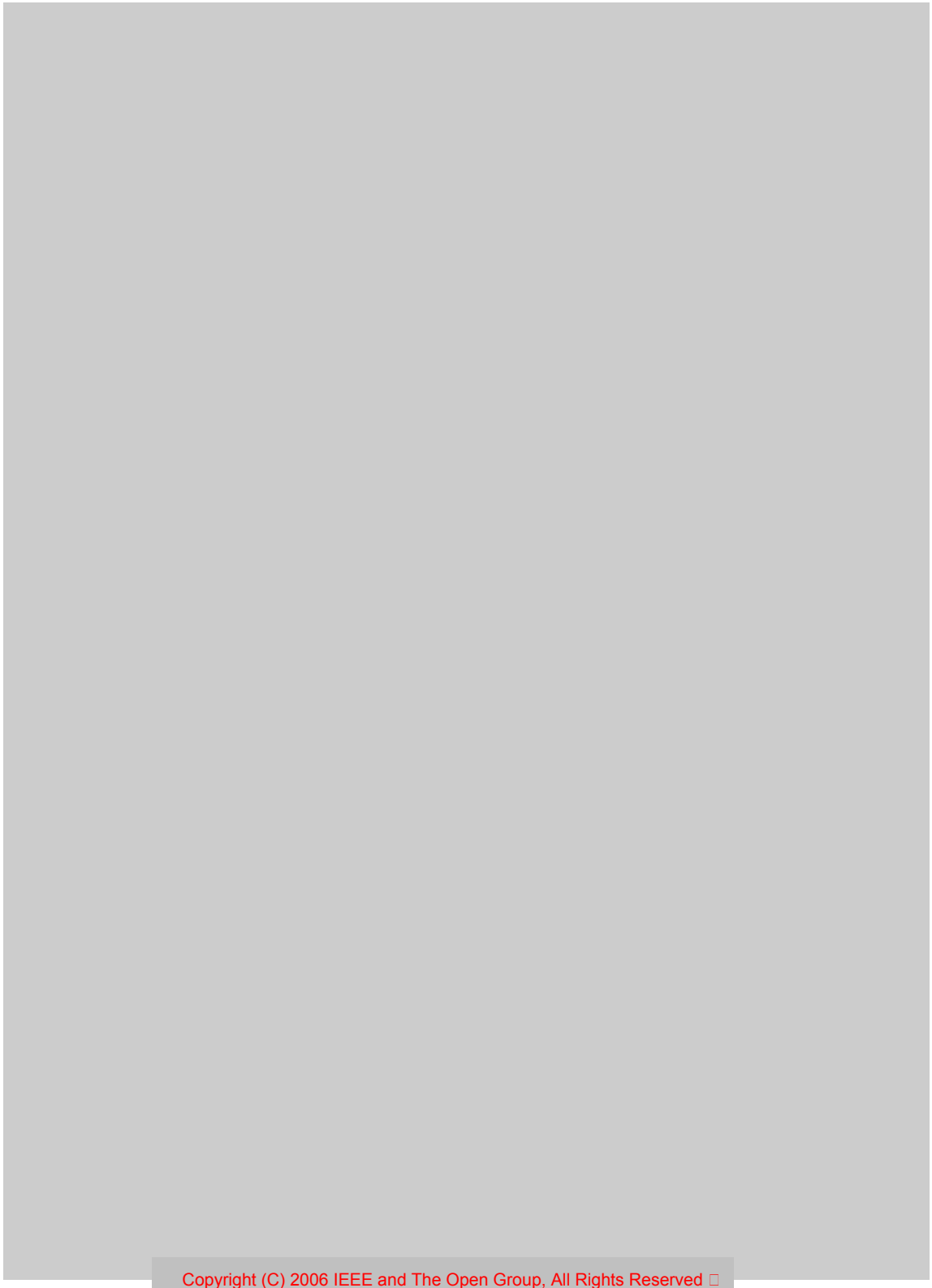
mbrlen — get number of bytes in a character (restartable)

SYNOPSIS

```
#include <wchar.h>

size_t mbrlen(const char *restrict s, size_t n,
    mbstate_t *restrict ps);
```

DESCRIPTION

CX



25463 **EXAMPLES**

25464 None.

25465 **APPLICATION USAGE**

25466 None.

25467 **RATIONALE**

25468 None.

25469 **FUTURE DIRECTIONS**

25470 None.

25471 **SEE ALSO**

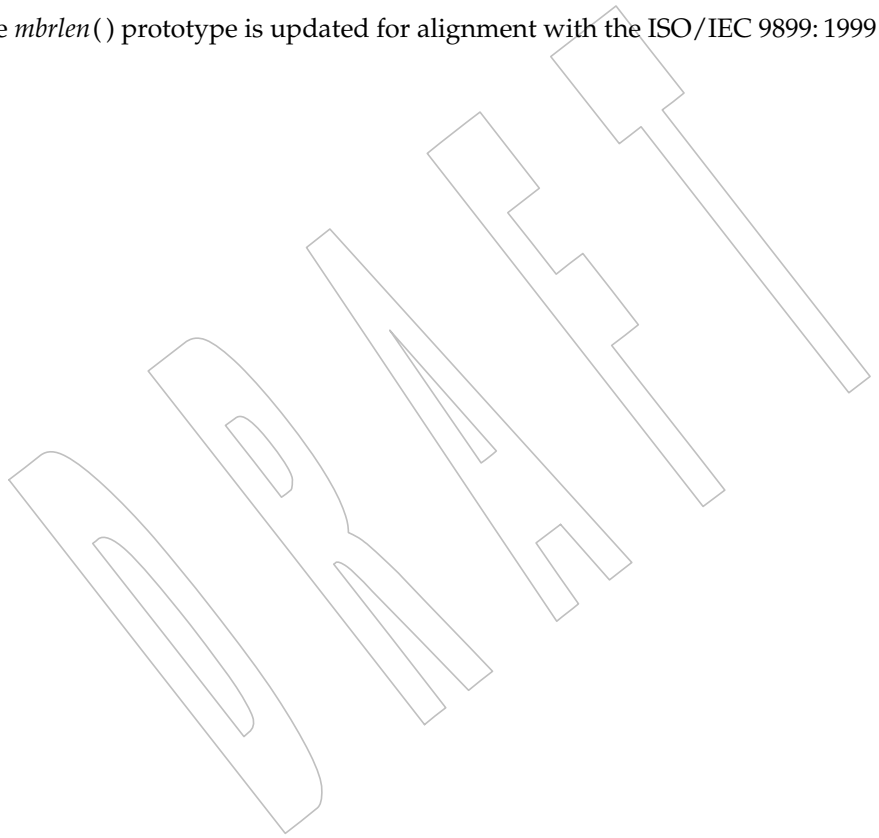
25472 *mbsinit*( ), *mbrtowc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

25473 **CHANGE HISTORY**

25474 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
25475 (E).

25476 **Issue 6**

25477 The *mbrlen*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.



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

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 1003.1-200x 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 1003.1-200x 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.

25520 **ERRORS**

25521 The *mbrtowc*( ) function may fail if:

25522 CX [EINVAL] *ps* points to an object that contains an invalid conversion state.

25523 [EILSEQ] Invalid character sequence is detected.

25524 **EXAMPLES**

25525 None.

25526 **APPLICATION USAGE**

25527 None.

25528 **RATIONALE**

25529 None.

25530 **FUTURE DIRECTIONS**

25531 None.

25532 **SEE ALSO**

25533 *mbsinit*( ), *mbsrtowcs*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

25534 **CHANGE HISTORY**

25535 First released in Issue 5. Included for alignment with ISO/IEC 9899: 1990/Amendment 1: 1995
25536 (E).

25537 **Issue 6**

25538 The *mbrtowc*( ) prototype is updated for alignment with the ISO/IEC 9899: 1999 standard.

25539 The following new requirements on POSIX implementations derive from alignment with the
25540 Single UNIX Specification:

25541 • The [EINVAL] error condition is added.

25542 ISO/IEC 9899: 1999 standard, Technical Corrigendum 1 is incorporated.

25543 **NAME**

25544 mbsinit — determine conversion object status

25545 **SYNOPSIS**

25546
```
#include <wchar.h>
```

25547
```
int mbsinit(const mbstate_t *ps);
```

25548 **DESCRIPTION**

25549 CX The functionality described on this reference page is aligned with the ISO C standard. Any
25550 conflict between the requirements described here and the ISO C standard is unintentional. This
25551 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

25552 If *ps* is not a null pointer, the *mbsinit*( ) function shall determine whether the object pointed to by
25553 *ps* describes an initial conversion state.

25554 **RETURN VALUE**

25555 The *mbsinit*( ) function shall return non-zero if *ps* is a null pointer, or if the pointed-to object
25556 describes an initial conversion state; otherwise, it shall return zero.

25557 If an **mbstate_t** object is altered by any of the functions described as ''restartable'', and is then
25558 used with a different character sequence, or in the other conversion direction, or with a different
25559 *LC_CTYPE* category setting than on earlier function calls, the behavior is undefined.

25560 **ERRORS**

25561 No errors are defined.

25562 **EXAMPLES**

25563 None.

25564 **APPLICATION USAGE**

25565 The **mbstate_t** object is used to describe the current conversion state from a particular character
25566 sequence to a wide-character sequence (or *vice versa*) under the rules of a particular setting of the
25567 *LC_CTYPE* category of the current locale.

25568 The initial conversion state corresponds, for a conversion in either direction, to the beginning of
25569 a new character sequence in the initial shift state. A zero valued **mbstate_t** object is at least one
25570 way to describe an initial conversion state. A zero valued **mbstate_t** object can be used to initiate
25571 conversion involving any character sequence, in any *LC_CTYPE* category setting.

25572 **RATIONALE**

25573 None.

25574 **FUTURE DIRECTIONS**

25575 None.

25576 **SEE ALSO**

25577 *mbrlen*( ), *mbrtowc*( ), *mbsrtowcs*( ), *wcrtomb*( ), *wcsrtombs*( ), the Base Definitions volume of
25578 IEEE Std 1003.1-200x, **<wchar.h>**

25579 **CHANGE HISTORY**

25580 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
25581 (E).

25582 **NAME**

25583 mbsnrtowcs — convert a multi-byte string to a wide-character string

25584 **SYNOPSIS**

25585 CX
```
#include <wchar.h>

size_t mbsnrtowcs(wchar_t *restrict dst, const char **restrict src,
    size_t nmc, size_t len, mbstate_t *restrict ps);
```

25588 **DESCRIPTION**

25589 Refer to *mbsrtowcs*( ).

NAME

mbsnrtowcs, mbsrtowcs — convert a character string to a wide-character string (restartable)

SYNOPSIS

```
#include <wchar.h>

CX  size_t mbsnrtowcs(wchar_t *restrict dst, const char **restrict src,
        size_t nmc, size_t len, mbstate_t *restrict ps);
size_t mbsrtowcs(wchar_t *restrict dst, const char **restrict src,
        size_t len, mbstate_t *restrict ps);
```

DESCRIPTION

CX For *mbsrtowcs*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

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 *mbrtowc*( ) 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.

CX The *mbsnrtowcs*( ) function shall be equivalent to the *mbsrtowcs*( ) function, except that the conversion of characters pointed to by *src* is limited to at most *nmc* bytes (the size of the input buffer).

The behavior of these functions shall be affected by the *LC_CTYPE* category of the current locale.

The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-200x calls these functions.

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, these functions shall store the value of the macro [EILSEQ] in *errno* and shall return (**size_t**)–1; the conversion state is undefined. Otherwise, these functions shall return the number of characters successfully converted, not including the terminating null (if any).

25632 **ERRORS**

25633 These functions may fail if:

25634 CX [EINVAL] *ps* points to an object that contains an invalid conversion state.

25635 [EILSEQ] Invalid character sequence is detected.

25636 **EXAMPLES**

25637 None.

25638 **APPLICATION USAGE**

25639 None.

25640 **RATIONALE**

25641 None.

25642 **FUTURE DIRECTIONS**

25643 None.

25644 **SEE ALSO**

25645 *iconv*( ), *mbrtowc*( ), *mbsinit*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

25646 **CHANGE HISTORY**

25647 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
25648 (E).

25649 **Issue 6**

25650 The *mbsrtowcs*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

25651 The [EINVAL] error condition is marked CX.

25652 **Issue 7**

25653 The *mbsnrtowcs*( ) function is added from The Open Group Technical Standard, 2006, Extended
25654 API Set Part 1.

25655 **NAME**

25656 mbstowcs — convert a character string to a wide-character string

25657 **SYNOPSIS**

25658
```
#include <stdlib.h>
```

25659
```
size_t mbstowcs(wchar_t *restrict pwcs, const char *restrict s,
    size_t n);
```

25661 **DESCRIPTION**

25662 CX The functionality described on this reference page is aligned with the ISO C standard. Any
25663 conflict between the requirements described here and the ISO C standard is unintentional. This
25664 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

25665 The *mbstowcs*( ) function shall convert a sequence of characters that begins in the initial shift
25666 state from the array pointed to by *s* into a sequence of corresponding wide-character codes and
25667 shall store not more than *n* wide-character codes into the array pointed to by *pwcs*. No
25668 characters that follow a null byte (which is converted into a wide-character code with value 0)
25669 shall be examined or converted. Each character shall be converted as if by a call to *mbtowc*( ),
25670 except that the shift state of *mbtowc*( ) is not affected.

25671 No more than *n* elements shall be modified in the array pointed to by *pwcs*. If copying takes
25672 place between objects that overlap, the behavior is undefined.

25673 XSI The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale.
25674 If *pwcs* is a null pointer, *mbstowcs*( ) shall return the length required to convert the entire array
25675 regardless of the value of *n*, but no values are stored.

25676 **RETURN VALUE**

25677 CX If an invalid character is encountered, *mbstowcs*( ) shall return (**size_t**)–1 and may set *errno* to
25678 indicate the error.

25679 XSI Otherwise, *mbstowcs*( ) shall return the number of the array elements modified (or required if
25680 *pwcs* is null), not including a terminating 0 code, if any. The array shall not be zero-terminated if
25681 the value returned is *n*.

25682 **ERRORS**

25683 The *mbstowcs*( ) function may fail if:

25684 XSI [EILSEQ] Invalid byte sequence is detected.

25685 **EXAMPLES**

25686 None.

25687 **APPLICATION USAGE**

25688 None.

25689 **RATIONALE**

25690 None.

25691 **FUTURE DIRECTIONS**

25692 None.

25693 **SEE ALSO**

25694 *mblen*( ), *mbtowc*( ), *wctomb*( ), *wcstombs*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
25695 **<stdlib.h>**

25696 **CHANGE HISTORY**

25697 First released in Issue 4. Aligned with the ISO C standard.

25698 **Issue 6**

25699 The *mbstowcs*( ) prototype is updated for alignment with the ISO/IEC 9899: 1999 standard.

25700 Extensions beyond the ISO C standard are marked.

25701 **NAME**

25702 mbtowc — convert a character to a wide-character code

25703 **SYNOPSIS**

25704 ```
#include <stdlib.h>
```

25705 ```
int mbtowc(wchar_t *restrict pwc, const char *restrict s, size_t n);
```

25706 **DESCRIPTION**

25707 CX The functionality described on this reference page is aligned with the ISO C standard. Any
25708 conflict between the requirements described here and the ISO C standard is unintentional. This
25709 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

25710 If *s* is not a null pointer, *mbtowc*( ) shall determine the number of bytes that constitute the
25711 character pointed to by *s*. It shall then determine the wide-character code for the value of type
25712 **wchar_t** that corresponds to that character. (The value of the wide-character code corresponding
25713 to the null byte is 0.) If the character is valid and *pwc* is not a null pointer, *mbtowc*( ) shall store
25714 the wide-character code in the object pointed to by *pwc*.

25715 The behavior of this function is affected by the *LC_CTYPE* category of the current locale. For a
25716 state-dependent encoding, this function is placed into its initial state by a call for which its
25717 character pointer argument, *s*, is a null pointer. Subsequent calls with *s* as other than a null
25718 pointer shall cause the internal state of the function to be altered as necessary. A call with *s* as a
25719 null pointer shall cause this function to return a non-zero value if encodings have state
25720 dependency, and 0 otherwise. If the implementation employs special bytes to change the shift
25721 state, these bytes shall not produce separate wide-character codes, but shall be grouped with an
25722 adjacent character. Changing the *LC_CTYPE* category causes the shift state of this function to be
25723 unspecified. At most *n* bytes of the array pointed to by *s* shall be examined.

25724 The implementation shall behave as if no function defined in this volume of
25725 IEEE Std 1003.1-200x calls *mbtowc*( ).

25726 **RETURN VALUE**

25727 If *s* is a null pointer, *mbtowc*( ) shall return a non-zero or 0 value, if character encodings,
25728 respectively, do or do not have state-dependent encodings. If *s* is not a null pointer, *mbtowc*( )
25729 shall either return 0 (if *s* points to the null byte), or return the number of bytes that constitute the
25730 CX converted character (if the next *n* or fewer bytes form a valid character), or return –1 and may
25731 set *errno* to indicate the error (if they do not form a valid character).

25732 In no case shall the value returned be greater than *n* or the value of the {MB_CUR_MAX} macro.

25733 **ERRORS**

25734 The *mbtowc*( ) function may fail if:

25735 XSI [EILSEQ] Invalid character sequence is detected.

25736 **EXAMPLES**

25737 None.

25738 **APPLICATION USAGE**

25739 None.

25740 **RATIONALE**

25741 None.

25742 **FUTURE DIRECTIONS**

25743 None.

25744 **SEE ALSO**

25745 *mblen*( ), *mbstowcs*( ), *wctomb*( ), *wcstombs*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
25746 **<stdlib.h>**

25747 **CHANGE HISTORY**

25748 First released in Issue 4. Aligned with the ISO C standard.

25749 **Issue 6**

25750 The *mbtowc*( ) prototype is updated for alignment with the ISO/IEC 9899: 1999 standard.

25751 Extensions beyond the ISO C standard are marked.

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 1003.1-200x, **<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.

25783 **NAME**

25784 memchr — find byte in memory

25785 **SYNOPSIS**

25786
```
#include <string.h>
```

25787
```
void *memchr(const void *s, int c, size_t n);
```

25788 **DESCRIPTION**

25789 CX The functionality described on this reference page is aligned with the ISO C standard. Any
25790 conflict between the requirements described here and the ISO C standard is unintentional. This
25791 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

25792 The *memchr*( ) function shall locate the first occurrence of *c* (converted to an **unsigned char**) in
25793 the initial *n* bytes (each interpreted as **unsigned char**) of the object pointed to by *s*.

25794 **RETURN VALUE**

25795 The *memchr*( ) function shall return a pointer to the located byte, or a null pointer if the byte does
25796 not occur in the object.

25797 **ERRORS**

25798 No errors are defined.

25799 **EXAMPLES**

25800 None.

25801 **APPLICATION USAGE**

25802 None.

25803 **RATIONALE**

25804 None.

25805 **FUTURE DIRECTIONS**

25806 None.

25807 **SEE ALSO**

25808 The Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

25809 **CHANGE HISTORY**

25810 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

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 1003.1-200x 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 1003.1-200x, **<string.h>**

CHANGE HISTORY

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

25842 **NAME**

25843 memcpy — copy bytes in memory

25844 **SYNOPSIS**

25845 ```
#include <string.h>
```

25846 ```
void *memcpy(void *restrict s1, const void *restrict s2, size_t n);
```

25847 **DESCRIPTION**

25848 CX The functionality described on this reference page is aligned with the ISO C standard. Any
25849 conflict between the requirements described here and the ISO C standard is unintentional. This
25850 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

25851 The *memcpy*( ) function shall copy *n* bytes from the object pointed to by *s2* into the object pointed
25852 to by *s1*. If copying takes place between objects that overlap, the behavior is undefined.

25853 **RETURN VALUE**

25854 The *memcpy*( ) function shall return *s1*; no return value is reserved to indicate an error.

25855 **ERRORS**

25856 No errors are defined.

25857 **EXAMPLES**

25858 None.

25859 **APPLICATION USAGE**

25860 The *memcpy*( ) function does not check for the overflow of the receiving memory area.

25861 **RATIONALE**

25862 None.

25863 **FUTURE DIRECTIONS**

25864 None.

25865 **SEE ALSO**

25866 The Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

25867 **CHANGE HISTORY**

25868 First released in Issue 1. Derived from Issue 1 of the SVID.

25869 **Issue 6**

25870 The *memcpy*( ) prototype is updated for alignment with the ISO/IEC 9899: 1999 standard.

25871 **NAME**

25872 memmove — copy bytes in memory with overlapping areas

25873 **SYNOPSIS**

25874 ```
#include <string.h>
```

25875 ```
void *memmove(void *s1, const void *s2, size_t n);
```

25876 **DESCRIPTION**

25877 CX The functionality described on this reference page is aligned with the ISO C standard. Any
25878 conflict between the requirements described here and the ISO C standard is unintentional. This
25879 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

25880 The *memmove*( ) function shall copy *n* bytes from the object pointed to by *s2* into the object
25881 pointed to by *s1*. Copying takes place as if the *n* bytes from the object pointed to by *s2* are first
25882 copied into a temporary array of *n* bytes that does not overlap the objects pointed to by *s1* and
25883 *s2*, and then the *n* bytes from the temporary array are copied into the object pointed to by *s1*.

25884 **RETURN VALUE**

25885 The *memmove*( ) function shall return *s1*; no return value is reserved to indicate an error.

25886 **ERRORS**

25887 No errors are defined.

25888 **EXAMPLES**

25889 None.

25890 **APPLICATION USAGE**

25891 None.

25892 **RATIONALE**

25893 None.

25894 **FUTURE DIRECTIONS**

25895 None.

25896 **SEE ALSO**

25897 The Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

25898 **CHANGE HISTORY**

25899 First released in Issue 4. Derived from the ANSI C standard.

25900 **NAME**

25901 memset — set bytes in memory

25902 **SYNOPSIS**

25903
```
#include <string.h>
```

25904
```
void *memset(void *s, int c, size_t n);
```

25905 **DESCRIPTION**

25906 CX The functionality described on this reference page is aligned with the ISO C standard. Any
25907 conflict between the requirements described here and the ISO C standard is unintentional. This
25908 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

25909 The *memset*( ) function shall copy *c* (converted to an **unsigned char**) into each of the first *n* bytes
25910 of the object pointed to by *s*.

25911 **RETURN VALUE**

25912 The *memset*( ) function shall return *s*; no return value is reserved to indicate an error.

25913 **ERRORS**

25914 No errors are defined.

25915 **EXAMPLES**

25916 None.

25917 **APPLICATION USAGE**

25918 None.

25919 **RATIONALE**

25920 None.

25921 **FUTURE DIRECTIONS**

25922 None.

25923 **SEE ALSO**

25924 The Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

25925 **CHANGE HISTORY**

25926 First released in Issue 1. Derived from Issue 1 of the SVID.

25927 **NAME**

25928 mkdir, mkdirat — make a directory relative to directory file descriptor

25929 **SYNOPSIS**

25930
```
#include <sys/stat.h>
```

25931
```
int mkdir(const char *path, mode_t mode);
```
25932
```
int mkdirat(int fd, const char *path, mode_t mode);
```

25933 **DESCRIPTION**

25934 The *mkdir*( ) function shall create a new directory with name *path*. The file permission bits of the
25935 new directory shall be initialized from *mode*. These file permission bits of the *mode* argument
25936 shall be modified by the process' file creation mask.

25937 When bits in *mode* other than the file permission bits are set, the meaning of these additional bits
25938 is implementation-defined.

25939 The directory's user ID shall be set to the process' effective user ID. The directory's group ID
25940 shall be set to the group ID of the parent directory or to the effective group ID of the process.
25941 Implementations shall provide a way to initialize the directory's group ID to the group ID of the
25942 parent directory. Implementations may, but need not, provide an implementation-defined way
25943 to initialize the directory's group ID to the effective group ID of the calling process.

25944 The newly created directory shall be an empty directory.

25945 If *path* names a symbolic link, *mkdir*( ) shall fail and set *errno* to [EEXIST].

25946 Upon successful completion, *mkdir*( ) shall mark for update the *st_atime*, *st_ctime*, and *st_mtime*
25947 fields of the directory. Also, the *st_ctime* and *st_mtime* fields of the directory that contains the
25948 new entry shall be marked for update.

25949 The *mkdirat*( ) function shall be equivalent to the *mkdir*( ) function except in the case where *path*
25950 specifies a relative path. In this case the newly created directory is created relative to the
25951 directory associated with the file descriptor *fd* instead of the current working directory. It is
25952 unspecified whether directory searches are permitted based on whether the file was opened
25953 with search permission or on the current permissions of the directory underlying the file
25954 descriptor.

25955 If *mkdirat*( ) is passed the special value AT_FDCWD in the *fd* parameter, the current working
25956 directory is used and the behavior shall be identical to a call to *mkdir*( ).

25957 **RETURN VALUE**

25958 Upon successful completion, these functions shall return 0. Otherwise, these functions shall
25959 return –1 and set *errno* to indicate the error. If –1 is returned, no directory shall be created.

25960 **ERRORS**

25961 These functions shall fail if:

25962 [EACCES] Search permission is denied on a component of the path prefix, or write
25963 permission is denied on the parent directory of the directory to be created.

25964 [EEXIST] The named file exists.

25965 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
25966 argument.

25967 [EMLINK] The link count of the parent directory would exceed {LINK_MAX}.

[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.

In addition, the *mkdirat*( ) function shall fail if:

[EBADF] The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

These functions 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}.

The *mkdirat*( ) function may fail if:

[ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

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.

The purpose of the *mkdirat*( ) function is to create a directory in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to the call to *mkdir*( ), resulting in unspecified behavior. By opening a file descriptor for the target directory and using the *mkdirat*( ) function it can be guaranteed that the newly created directory is located relative to the desired directory.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod( ), *mkdtemp*( ), *mknod*( ), *umask*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

Issue 7

The *mkdirat*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 2.

26038 **NAME**

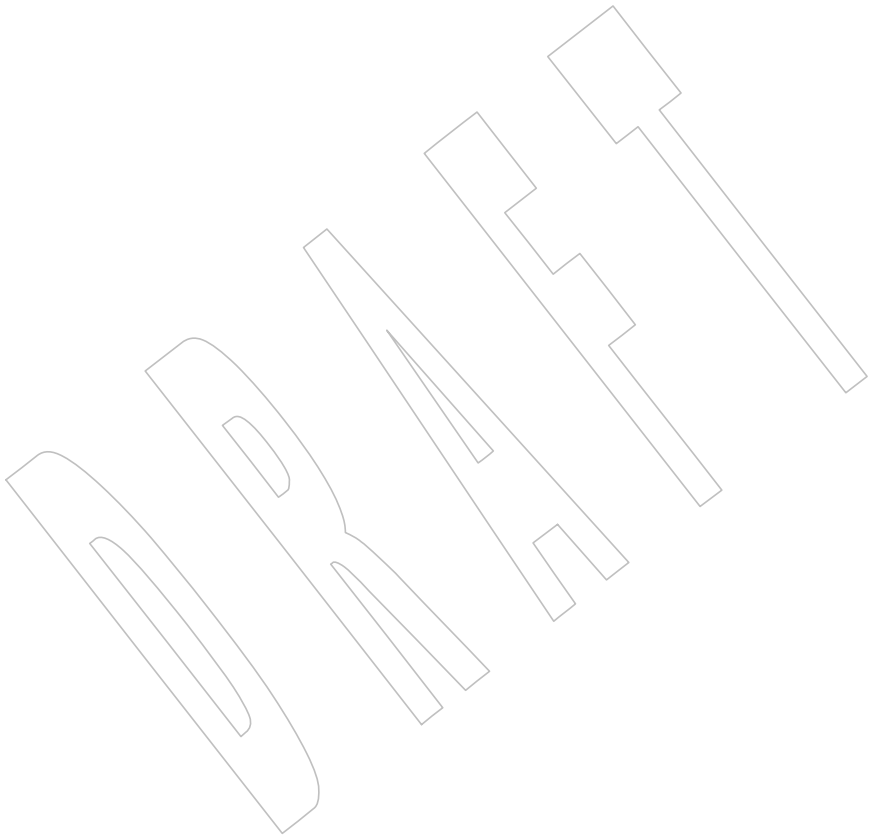
26039 mkdirat — make a directory relative to directory file descriptor

26040 **SYNOPSIS**

26041 ```
#include <sys/stat.h>
```

26042 ```
int mkdirat(int fd, const char *path, mode_t mode);
```

26043 **DESCRIPTION**

26044 Refer to *mkdir( )*.



NAME

mkdtemp, mkstemp — create a unique directory or file

SYNOPSIS

CX
```
#include <stdlib.h>

char *mkdtemp(char *template);
int mkstemp(char *template);
```

DESCRIPTION

The *mkdtemp*( ) function uses the contents of *template* to construct a unique directory name. The string provided in *template* shall be a filename ending with six trailing `'X'`s. The *mkdtemp*( ) function shall replace 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 *mkdtemp*( ). The unique directory name is used to attempt to create the directory using mode 0700 as modified by the file creation mask.

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 *mkstemp*( ) function shall create the file, and obtain a file descriptor for it, as if by a call to:

```
open(filename, O_RDWR|O_CREAT|O_EXCL, S_IRUSR|S_IWUSR)
```

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, the *mkdtemp*( ) function shall return a pointer to the string containing the directory name if it was created. Otherwise, it shall return a null pointer and shall set *errno* to indicate the error.

Upon successful completion, *mkstemp*( ) shall return an open file descriptor. Otherwise, –1 shall be returned if no suitable file could be created.

ERRORS

These functions 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.

[EINVAL] The string pointed to by *template* does not end in `"XXXXXX"`.

[ELOOP] A loop exists in symbolic links encountered during resolution of the path of the directory to be created.

[EMLINK] The link count of the parent directory would exceed {LINK_MAX}.

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

[ENOENT] A component of the path prefix specified by the *template* argument 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.

These functions may fail if:

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the path of the directory to be created.

[ENAMETOOLONG]
As a result of encountering a symbolic link in resolution of the path of the directory to be created, the length of the substituted pathname string exceeded {PATH_MAX}.

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 *mkdtemp*( ) and *mkstemp*( ) functions 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( ), *mkdir*( ), *open*( ), *tmpfile*( ), *tmpnam*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 7

The *mkstemp*( ) function is moved from the XSI option to the Base.

The *mkdtemp*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 1.

SD5-XSH-ERN-168 is applied, clarifying file permissions upon creation.

NAME

mkfifo, mkfifoat — make a FIFO special file relative to directory file descriptor

SYNOPSIS

```
#include <sys/stat.h>

int mkfifo(const char *path, mode_t mode);
int mkfifoat(int fd, 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.

The *mkfifoat*( ) function shall be equivalent to the *mkfifo*( ) function except in the case where *path* specifies a relative path. In this case the newly created FIFO is created relative to the directory associated with the file descriptor *fd* instead of the current working directory. It is unspecified whether directory searches are permitted based on whether the file was opened with search permission or on the current permissions of the directory underlying the file descriptor.

If *mkfifoat*( ) is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior shall be identical to a call to *mkfifo*( ).

RETURN VALUE

Upon successful completion, these functions shall return 0. Otherwise, these functions shall return −1 and set *errno* to indicate the error. If −1 is returned, no FIFO shall be created.

ERRORS

These functions 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 *mkfifoat*( ) function shall fail if:

[EBADF] The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

These functions 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}.

The *mkfifoat*( ) function may fail if:

[ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

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.

```
#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*( ). 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.

The purpose of the *mkfifoat*( ) function is to create a FIFO special file in directories other than the current working directory without exposure to race conditions. Any part of the path of a file

could be changed in parallel to a call to *mkfifo*( ), resulting in unspecified behavior. By opening a file descriptor for the target directory and using the *mkfifoat*( ) function it can be guaranteed that the newly created FIFO is located relative to the desired directory.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod( ), *mknod*( ), *umask*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

Issue 7

The *mkfifoat*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 2.

26239 **NAME**

26240 mkfifoat — make a FIFO special file relative to directory file descriptor

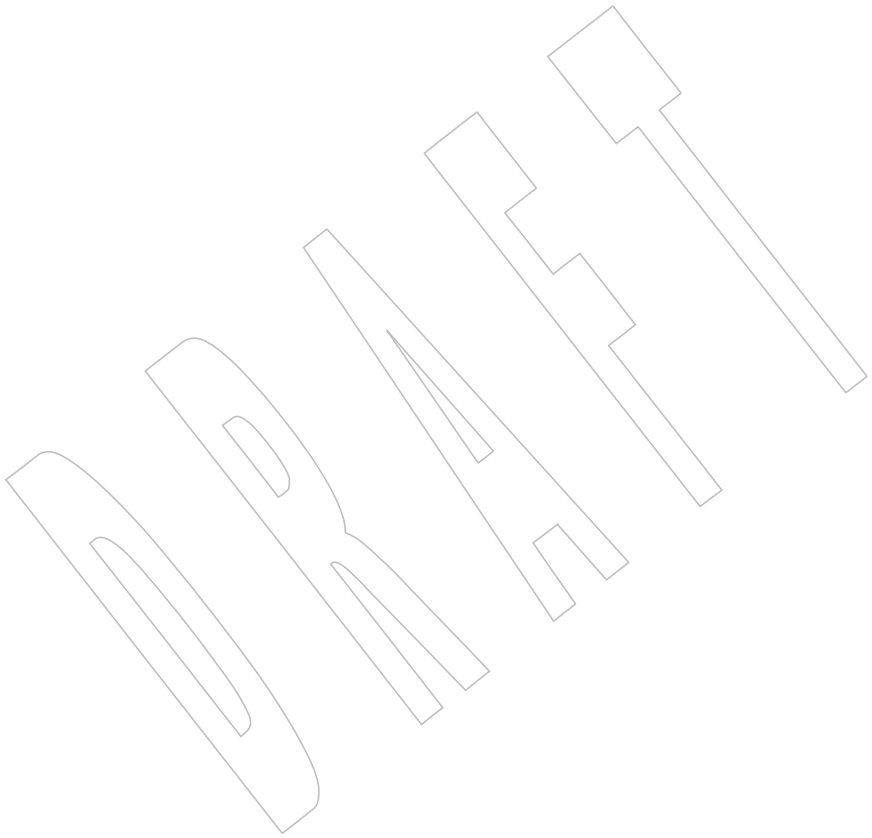
26241 **SYNOPSIS**

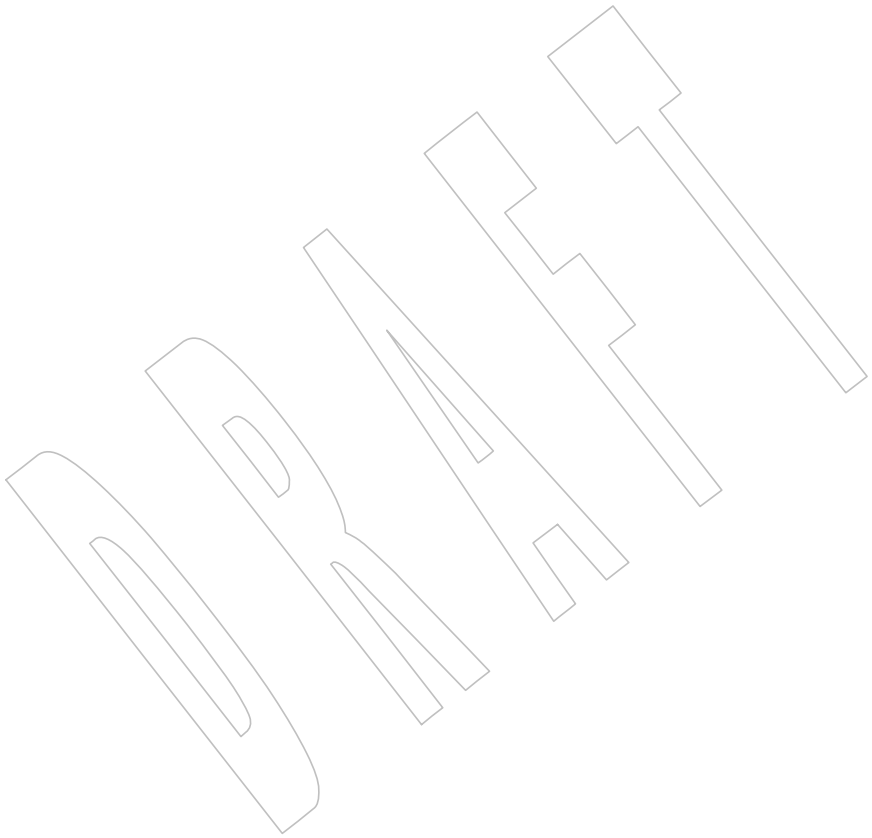
26242
```
#include <sys/stat.h>
```

26243
```
int mkfifoat(int fd, const char *path, mode_t mode);
```

26244 **DESCRIPTION**

26245 Refer to *mkfifo( )*.





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.

The *mknodat*( ) function shall be equivalent to the *mknod*( ) function except in the case where *path* specifies a relative path. In this case the newly created directory, special file, or regular file is located relative to the directory associated with the file descriptor *fd* instead of the current working directory. It is unspecified whether directory searches are permitted based on whether the file was opened with search permission or on the current permissions of the directory underlying the file descriptor.

If *mknodat*( ) is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior shall be identical to a call to *mknod*( ).

RETURN VALUE

Upon successful completion, these functions shall return 0. Otherwise, these functions shall return –1 and set *errno* to indicate the error. If –1 is returned, the new file shall not be created.

ERRORS

These functions 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.

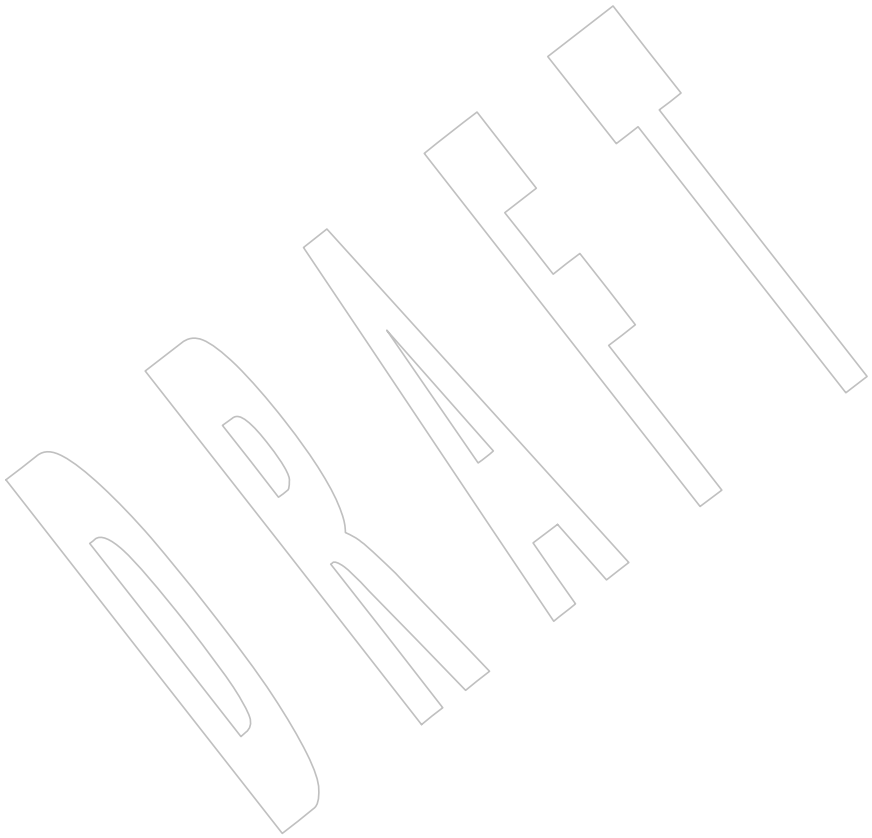
[EINVAL] An invalid argument exists.

[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]

[ELOOP] More t



Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The normative text 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.

Issue 7

The *mknodat*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 2.

26385 **NAME**

26386 mknodat — make directory, special file, or regular file

26387 **SYNOPSIS**

26388 XSI
```
#include <sys/stat.h>

int mknodat(int fd, const char *path, mode_t mode, dev_t dev);
```

26390 **DESCRIPTION**

26391 Refer to *mknod( )*.

26392 **NAME**

26393 mkstemp — create a unique directory

26394 **SYNOPSIS**

26395 CX
```
#include <stdlib.h>

int mkstemp(char *template);
```
26396

26397 **DESCRIPTION**

26398 Refer to *mkdtemp( )*.

NAME

mktime — convert broken-down time into time since the Epoch

SYNOPSIS

```
#include <time.h>

time_t mktime(struct tm *timeptr);
```

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 1003.1-200x defers to the ISO C standard.

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>**.

CX 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 1003.1-200x, 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 CX and may set *errno* to indicate the error.

ERRORS

The *mktime*( ) function may fail if:

CX [EOVERFLOW] The result cannot be represented.

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;
        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(daybuf, sizeof(daybuf), "%A", &time_str);
            (void)puts(daybuf);
        }
        return 0;
    }
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

asctime( ), *clock*( ), *ctime*( ), *difftime*( ), *gmtime*( ), *localtime*( ), *strftime*( ), *strptime*( ), *time*( ), *tzset*( ), *utime*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard and the ANSI C standard.

Issue 6

Extensions beyond the ISO C standard are marked.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/58 is applied, updating the RETURN VALUE and ERRORS sections to add the optional [EOVERFLOW] error as a CX extension.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/59 is applied, adding the *tzset*( ) function to the SEE ALSO section.

NAME

mlock, munlock — lock or unlock a range of process address space (**REALTIME**)

SYNOPSIS

MLR
```
#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 *exec*s 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.

[EPERM] The calling process does not have the appr

NAME

mlockall, munlockall — lock/unlock the address space of a process (**REALTIME**)

SYNOPSIS

ML

```
#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 *exec*s 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 address space of the process shall be memory-resident and locked. Upon return from the *munlockall*( ) function, all currently mapped pages of the address space of the process shall be unlocked with respect to the address space of the process. 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.

[EINVAL] The *flags* argument is zero, or includes unimplemented flags.

The *mlockall*( ) function may fail if:

[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 1003.1-200x, **<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

```
#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 an address space of a process and a memory object.

The *mmap*( ) function shall be supported for the following memory objects:

- Regular files
- SHM Shared memory objects
- TYM Typed memory objects

Support for any other type of file is unspecified.

The format of the call is as follows:

```
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 memory object represented by *fildes*.

TYM 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.

If *len* is zero, *mmap*( ) shall fail and no mapping shall be established.

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.

Symbolic Constant	Description
PROT_READ	Data can be read.
PROT_WRITE	Data can be written.
PROT_EXEC	Data can be executed.
PROT_NONE	Data cannot be accessed.

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, the implementation shall not permit a write to succeed where PROT_WRITE has not been set and 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. 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>**:

Symbolic Constant	Description
MAP_SHARED	Changes are shared.
MAP_PRIVATE	Changes are private.
MAP_FIXED	Interpret *addr* exactly.

XSI 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*( ).

The state of synchronization objects such as mutexes, semaphores, barriers, and conditional variables placed in shared memory mapped with MAP_SHARED becomes undefined when the last region in any process containing the synchronization object is unmapped.

TYM 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 pages in the range [*pa*,*pa*+*len*) of the process.

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.

If MAP_FIXED is specified and *addr* is non-zero, it shall have the same remainder as the *off* parameter, modulo the page size as returned by *sysconf*( ) when passed _SC_PAGESIZE or _SC_PAGE_SIZE. The implementation may require that off is a multiple of the page size. If MAP_FIXED is specified, the implementation may require that *addr* is a multiple of the page size. The system performs mapping operations over whole pages. Thus, while the parameter *len* need not meet a size or alignment constraint, the system shall include, in any mapping operation, any partial page specified by the address range starting at *pa* and continuing for *len* bytes.

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).

XSI 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.

26742 **RETURN VALUE**

26743 Upon successful completion, the *mmap*( ) function shall return the address at which the mapping
26744 was placed (*pa*); otherwise, it shall return a value of MAP_FAILED and set *errno* to indicate the
26745 error. The symbol MAP_FAILED is defined in the **<sys/mman.h>** header. No successful return
26746 from *mmap*( ) shall return the value MAP_FAILED.

26747 **ERRORS**

26748 The *mmap*( ) function shall fail if:

26749 [EACCES] The *fildes* argument is not open for read, regardless of the protection specified,
26750 or *fildes* is not open for write and PROT_WRITE was specified for a
26751 MAP_SHARED type mapping.

26752 ML [EAGAIN] The mapping could not be locked in memory, if required by *mlockall*( ), due to
26753 a lack of resources.

26754 [EBADF] The *fildes* argument is not a valid open file descriptor.

26755 [EINVAL] The value of *len* is zero.

26756 [EINVAL] The value of *flags* is invalid (neither MAP_PRIVATE nor MAP_SHARED is
26757 set).

26758 [EMFILE] The number of mapped regions would exceed an implementation-defined
26759 limit (per process or per system).

26760 [ENODEV] The *fildes* argument refers to a file whose type is not supported by *mmap*( ).

26761 [ENOMEM] MAP_FIXED was specified, and the range [*addr*,*addr*+*len*) exceeds that allowed
26762 for the address space of a process; or, if MAP_FIXED was not specified and
26763 there is insufficient room in the address space to effect the mapping.

26764 ML [ENOMEM] The mapping could not be locked in memory, if required by *mlockall*( ),
26765 because it would require more space than the system is able to supply.

26766 TYM [ENOMEM] Not enough unallocated memory resources remain in the typed memory
26767 object designated by *fildes* to allocate *len* bytes.

26768 [ENOTSUP] MAP_FIXED or MAP_PRIVATE was specified in the *flags* argument and the
26769 implementation does not support this functionality.

26770 The implementation does not support the combination of accesses requested
26771 in the *prot* argument.

26772 [ENXIO] Addresses in the range [*off*,*off*+*len*) are invalid for the object specified by *fildes*.

26773 [ENXIO] MAP_FIXED was specified in *flags* and the combination of *addr*, *len*, and *off* is
26774 invalid for the object specified by *fildes*.

26775 TYM [ENXIO] The *fildes* argument refers to a typed memory object that is not accessible from
26776 the calling process.

26777 [EOVERFLOW] The file is a regular file and the value of *off* plus *len* exceeds the offset
26778 maximum established in the open file description associated with *fildes*.

26779 The *mmap*( ) function may fail if:

26780 [EINVAL] The *addr* argument (if MAP_FIXED was specified) or *off* is not a multiple of the
26781 page size as returned by *sysconf*( ), or is considered invalid by the
26782 implementation.

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. Most implementations require that *off* and *addr* are multiples of the page size as returned by *sysconf*( ).

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:

```
fildes = open(...)
lseek(fildes, some_offset)
read(fildes, buf, len)
/* Use data in buf. */
```

becomes:

```
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 1003.1-200x specifies that the new mappings replace any existing mappings, following existing practice in this regard.

It is not expected that all hardware implementations are able to support all combinations of

26831 permissions at all addresses. Implementations are required to disallow write access to mappings
26832 without write permission and to disallow access to mappings without any access permission.
26833 Other than these restrictions, implementations may allow access types other than those
26834 requested by the application. For example, if the application requests only PROT_WRITE, the
26835 implementation may also allow read access. A call to *mmap*( ) fails if the implementation cannot
26836 support allowing all the access requested by the application. For example, some
26837 implementations cannot support a request for both write access and execute access
26838 simultaneously. All implementations must support requests for no access, read access, write
26839 access, and both read and write access. Strictly conforming code must only rely on the required
26840 checks. These restrictions allow for portability across a wide range of hardware.

26841 The MAP_FIXED address treatment is likely to fail for non-page-aligned values and for certain
26842 architecture-dependent address ranges. Conforming implementations cannot count on being
26843 able to choose address values for MAP_FIXED without utilizing non-portable, implementation-
26844 defined knowledge. Nonetheless, MAP_FIXED is provided as a standard interface conforming
26845 to existing practice for utilizing such knowledge when it is available.

26846 Similarly, in order to allow implementations that do not support virtual addresses, support for
26847 directly specifying any mapping addresses via MAP_FIXED is not required and thus a
26848 conforming application may not count on it.

26849 The MAP_PRIVATE function can be implemented efficiently when memory protection hardware
26850 is available. When such hardware is not available, implementations can implement such
26851 "mappings" by simply making a real copy of the relevant data into process private memory,
26852 though this tends to behave similarly to *read*( ).

26853 The function has been defined to allow for many different models of using shared memory.
26854 However, all uses are not equally portable across all machine architectures. In particular, the
26855 *mmap*( ) function allows the system as well as the application to specify the address at which to
26856 map a specific region of a memory object. The most portable way to use the function is always to
26857 let the system choose the address, specifying NULL as the value for the argument *addr* and not
26858 to specify MAP_FIXED.

26859 If it is intended that a particular region of a memory object be mapped at the same address in a
26860 group of processes (on machines where this is even possible), then MAP_FIXED can be used to
26861 pass in the desired mapping address. The system can still be used to choose the desired address
26862 if the first such mapping is made without specifying MAP_FIXED, and then the resulting
26863 mapping address can be passed to subsequent processes for them to pass in via MAP_FIXED.
26864 The availability of a specific address range cannot be guaranteed, in general.

26865 The *mmap*( ) function can be used to map a region of memory that is larger than the current size
26866 of the object. Memory access within the mapping but beyond the current end of the underlying
26867 objects may result in SIGBUS signals being sent to the process. The reason for this is that the size
26868 of the object can be manipulated by other processes and can change at any moment. The
26869 implementation should tell the application that a memory reference is outside the object where
26870 this can be detected; otherwise, written data may be lost and read data may not reflect actual
26871 data in the object.

26872 Note that references beyond the end of the object do not extend the object as the new end cannot
26873 be determined precisely by most virtual memory hardware. Instead, the size can be directly
26874 manipulated by *ftruncate*( ).

26875 Process memory locking does apply to shared memory regions, and the MEMLOCK_FUTURE
26876 argument to *mlockall*( ) can be relied upon to cause new shared memory regions to be
26877 automatically locked.

26878 Existing implementations of *mmap*( ) return the value –1 when unsuccessful. Since the casting of
26879 this value to type **void** * cannot be guaranteed by the ISO C standard to be distinct from a
26880 successful value, this volume of IEEE Std 1003.1-200x defines the symbol MAP_FAILED, which a

conforming implementation does not return as the result of a successful call.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *fcntl( )*, *fork( )*, *lockf( )*, *msync( )*, *munmap( )*, *mprotect( )*, *posix_typed_mem_open( )*, *shmat( )*, *sysconf( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/mman.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

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 describe 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 normative text is updated to avoid use of the term "must" for application requirements.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/34 is applied, changing the margin code in the SYNOPSIS from MF | SHM to MC3 (notation for MF | SHM | TYM).

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/60 is applied, updating the DESCRIPTION and ERRORS sections to add the [EINVAL] error when *len* is zero.

Issue 7

Austin Group Interpretations 1003.1-2001 #078 and #079 are applied, clarifying page alignment requirements and adding a note about the state of synchronization objects becoming undefined when a shared region is unmapped.

Functionality relating to the Memory Protection and Memory Mapped Files options is moved to the Base.

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

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 1003.1-200x defers to the ISO C standard.

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 1003.1-200x, **<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.

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

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/35 is applied, correcting the code example in the APPLICATION USAGE section.

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 may 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.

[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.

[ENOTSUP] The implementation does not support the combination of accesses requested in the *prot* argument.

The *mprotect*( ) function may fail if:

[EINVAL] The *addr* argument is not a multiple of the page size as returned by *sysconf*( ).

EXAMPLES

None.

APPLICATION USAGE

Most implementations require that *addr* is a multiple of the page size as returned by *sysconf*( ).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mmap( ), *sysconf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/mman.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Aligned with *mprotect*( ) in the POSIX Realtime Extension as follows:

- The DESCRIPTION is largely reworded.
- [ENOTSUP] and a second form of [ENOMEM] are added as mandatory error conditions.
- [EAGAIN] is moved from the optional to the mandatory error conditions.

Issue 6

The *mprotect*( ) function is marked as part of the Memory Protection 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 as returned by *sysconf*( ).
- The [EINVAL] error condition is added.

The normative text is updated to avoid use of the term "must" for application requirements.

Issue 7

SD5-XSH-ERN-22 is applied, deleting erroneous APPLICATION USAGE.

Austin Group Interpretation 1003.1-2001 #078 is applied, clarifying page alignment requirements.

The *mprotect*( ) function is moved from the Memory Protection option to the Base.

27056 **NAME**

27057 mq_close — close a message queue (**REALTIME**)

27058 **SYNOPSIS**

27059 MSG
```
#include <mqueue.h>

int mq_close(mqd_t mqdes);
```

27061 **DESCRIPTION**

27062 The *mq_close*( ) function shall remove the association between the message queue descriptor,
27063 *mqdes*, and its message queue. The results of using this message queue descriptor after successful
27064 return from this *mq_close*( ), and until the return of this message queue descriptor from a
27065 subsequent *mq_open*( ), are undefined.

27066 If the process has successfully attached a notification request to the message queue via this
27067 *mqdes*, this attachment shall be removed, and the message queue is available for another process
27068 to attach for notification.

27069 **RETURN VALUE**

27070 Upon successful completion, the *mq_close*( ) function shall return a value of zero; otherwise, the
27071 function shall return a value of –1 and set *errno* to indicate the error.

27072 **ERRORS**

27073 The *mq_close*( ) function shall fail if:

27074 [EBADF] The *mqdes* argument is not a valid message queue descriptor.

27075 **EXAMPLES**

27076 None.

27077 **APPLICATION USAGE**

27078 None.

27079 **RATIONALE**

27080 None.

27081 **FUTURE DIRECTIONS**

27082 None.

27083 **SEE ALSO**

27084 *mq_open*( ), *mq_unlink*( ), *msgctl*( ), *msgget*( ), *msgrcv*( ), *msgsnd*( ), the Base Definitions volume of
27085 IEEE Std 1003.1-200x, **<mqueue.h>**

27086 **CHANGE HISTORY**

27087 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

27088 **Issue 6**

27089 The *mq_close*( ) function is marked as part of the Message Passing option.

27090 The [ENOSYS] error condition has been removed as stubs need not be provided if an
27091 implementation does not support the Message Passing option.

27092 **NAME**

27093 mq_getattr — get message queue attributes (**REALTIME**)

27094 **SYNOPSIS**

27095 MSG
```
#include <mqueue.h>
```
27096
```
int mq_getattr(mqd_t mqdes, struct mq_attr *mqstat);
```

27097 **DESCRIPTION**

27098 The *mq_getattr*( ) function shall obtain status information and attributes of the message queue
27099 and the open message queue description associated with the message queue descriptor.

27100 The *mqdes* argument specifies a message queue descriptor.

27101 The results shall be returned in the **mq_attr** structure referenced by the *mqstat* argument.

27102 Upon return, the following members shall have the values associated with the open message
27103 queue description as set when the message queue was opened and as modified by subsequent
27104 *mq_setattr*( ) calls: *mq_flags*.

27105 The following attributes of the message queue shall be returned as set at message queue
27106 creation: *mq_maxmsg*, *mq_msgsize*.

27107 Upon return, the following members within the **mq_attr** structure referenced by the *mqstat*
27108 argument shall be set to the current state of the message queue:

27109 *mq_curmsgs* The number of messages currently on the queue.

27110 **RETURN VALUE**

27111 Upon successful completion, the *mq_getattr*( ) function shall return zero. Otherwise, the function
27112 shall return –1 and set *errno* to indicate the error.

27113 **ERRORS**

27114 The *mq_getattr*( ) function may fail if:

27115 [EBADF] The *mqdes* argument is not a valid message queue descriptor.

27116 **EXAMPLES**

27117 None.

27118 **APPLICATION USAGE**

27119 None.

27120 **RATIONALE**

27121 None.

27122 **FUTURE DIRECTIONS**

27123 None.

27124 **SEE ALSO**

27125 *mq_open*( ), *mq_send*( ), *mq_setattr*( ), *mq_timedsend*( ), *msgctl*( ), *msgget*( ), *msgrcv*( ), *msgsnd*( ), the
27126 Base Definitions volume of IEEE Std 1003.1-200x, **<mqueue.h>**

27127 **CHANGE HISTORY**

27128 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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/61 is applied, updating the ERRORS section to change the [EBADF] error from mandatory to optional.

27137 **NAME**

27138 mq_notify — notify process that a message is available (**REALTIME**)

27139 **SYNOPSIS**

27140 MSG
```
#include <mqueue.h>

int mq_notify(mqd_t mqdes, const struct sigevent *notification);
```

27142 **DESCRIPTION**

27143 If the argument *notification* is not NULL, this function shall register the calling process to be
27144 notified of message arrival at an empty message queue associated with the specified message
27145 queue descriptor, *mqdes*. The notification specified by the *notification* argument shall be sent to
27146 the process when the message queue transitions from empty to non-empty. At any time, only
27147 one process may be registered for notification by a message queue. If the calling process or any
27148 other process has already registered for notification of message arrival at the specified message
27149 queue, subsequent attempts to register for that message queue shall fail.

27150 If *notification* is NULL and the process is currently registered for notification by the specified
27151 message queue, the existing registration shall be removed.

27152 When the notification is sent to the registered process, its registration shall be removed. The
27153 message queue shall then be available for registration.

27154 If a process has registered for notification of message arrival at a message queue and some
27155 thread is blocked in *mq_receive*( ) or *mq_timedreceive*( ) waiting to receive a message when a
27156 message arrives at the queue, the arriving message shall satisfy the appropriate *mq_receive*( ) or
27157 *mq_timedreceive*( ), respectively. The resulting behavior is as if the message queue remains empty,
27158 and no notification shall be sent.

27159 **RETURN VALUE**

27160 Upon successful completion, the *mq_notify*( ) function shall return a value of zero; otherwise, the
27161 function shall return a value of –1 and set *errno* to indicate the error.

27162 **ERRORS**

27163 The *mq_notify*( ) function shall fail if:

27164 [EBADF] The *mqdes* argument is not a valid message queue descriptor.

27165 [EBUSY] A process is already registered for notification by the message queue.

27166 The *mq_notify*( ) function may fail if:

27167 [EINVAL] The *notification* argument is NULL and the process is currently not registered.

27168 **EXAMPLES**

27169 None.

27170 **APPLICATION USAGE**

27171 None.

27172 **RATIONALE**

27173 None.

27174 **FUTURE DIRECTIONS**

27175 None.

SEE ALSO

mq_open( ), *mq_send( )*, *mq_receive( )*, *mq_timedsend( )*, *msgctl( )*, *msgget( )*, *msgrcv( )*, *msgsnd( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<mqueue.h>**

CHANGE HISTORY

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

Issue 6

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.

Issue 7

SD5-XSH-ERN-38 is applied, adding the *mq_timedreceive( )* function to the DESCRIPTION.

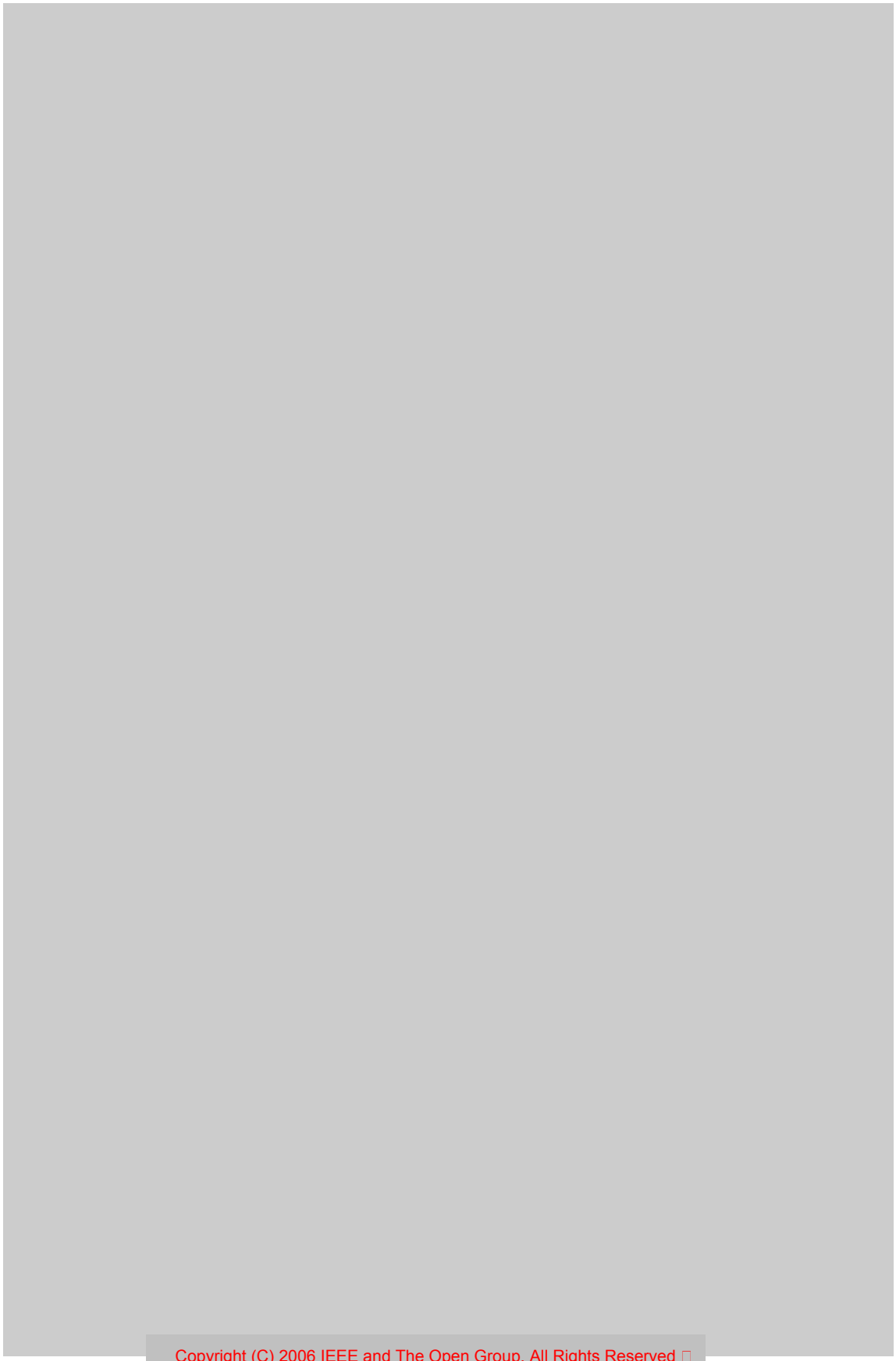
Austin Group Interpretation 1003.1-2001 #032 is applied, adding the [EINVAL] error.

NAME

mq_open — open a message queue (**REALTIME**)

SYNOPSIS

MSG



27235 O_EXCL. Otherwise, a message queue shall be created without any messages
27236 in it. The user ID of the message queue shall be set to the effective user ID of
27237 the process, and the group ID of the message queue shall be set to the effective
27238 group ID of the process. The permission bits of the message queue shall be set
27239 to the value of the *mode* argument, except those set in the file mode creation
27240 mask of the process. When bits in *mode* other than the file permission bits are
27241 specified, the effect is unspecified. If *attr* is NULL, the message queue shall be
27242 created with implementation-defined default message queue attributes. If *attr*
27243 is non-NULL and the calling process has the appropriate privilege on *name*,
27244 the message queue *mq_maxmsg* and *mq_msgsize* attributes shall be set to the
27245 values of the corresponding members in the **mq_attr** structure referred to by
27246 *attr*. If *attr* is non-NULL, but the calling process does not have the appropriate
27247 privilege on *name*, the *mq_open*( ) function shall fail and return an error
27248 without creating the message queue.

27249 O_EXCL If O_EXCL and O_CREAT are set, *mq_open*( ) shall fail if the message queue
27250 *name* exists. The check for the existence of the message queue and the creation
27251 of the message queue if it does not exist shall be atomic with respect to other
27252 threads executing *mq_open*( ) naming the same *name* with O_EXCL and
27253 O_CREAT set. If O_EXCL is set and O_CREAT is not set, the result is
27254 undefined.

27255 O_NONBLOCK Determines whether an *mq_send*( ) or *mq_receive*( ) waits for resources or
27256 messages that are not currently available, or fails with *errno* set to [EAGAIN];
27257 see *mq_send*( ) and *mq_receive*( ) for details.

27258 The *mq_open*( ) function does not add or remove messages from the queue.

27259 **RETURN VALUE**

27260 Upon successful completion, the function shall return a message queue descriptor; otherwise,
27261 the function shall return (**mqd_t**)–1 and set *errno* to indicate the error.

27262 **ERRORS**

27263 The *mq_open*( ) function shall fail if:

27264 [EACCES] The message queue exists and the permissions specified by *oflag* are denied, or
27265 the message queue does not exist and permission to create the message queue
27266 is denied.

27267 [EEXIST] O_CREAT and O_EXCL are set and the named message queue already exists.

27268 [EINTR] The *mq_open*( ) function was interrupted by a signal.

27269 [EINVAL] The *mq_open*( ) function is not supported for the given name.

27270 [EINVAL] O_CREAT was specified in *oflag*, the value of *attr* is not NULL, and either
27271 *mq_maxmsg* or *mq_msgsize* was less than or equal to zero.

27272 [EMFILE] Too many message queue descriptors or file descriptors are currently in use by
27273 this process.

27274 [ENFILE] Too many message queues are currently open in the system.

27275 [ENOENT] O_CREAT is not set and the named message queue does not exist.

27276 [ENOSPC] There is insufficient space for the creation of the new message queue.

27277 If any of the following conditions occur, the *mq_open*( ) function may return (**mqd_t**)–1 and set
27278 *errno* to the corresponding value.

27279 [ENAMETOOLONG]
27280 The length of the *name* argument exceeds {_POSIX_PATH_MAX} on systems
27281 XSI that do not support the XSI option or exceeds {_XOPEN_PATH_MAX} on XSI
27282 systems, or has a pathname component that is longer than
27283 XSI {_POSIX_NAME_MAX} on systems that do not support the XSI option or
27284 longer than {_XOPEN_NAME_MAX} on XSI systems.

27285 **EXAMPLES**
27286 None.

27287 **APPLICATION USAGE**
27288 None.

27289 **RATIONALE**
27290 None.

27291 **FUTURE DIRECTIONS**
27292 None.

27293 **SEE ALSO**
27294 *mq_close( )*, *mq_getattr( )*, *mq_receive( )*, *mq_send( )*, *mq_setattr( )*, *mq_timedreceive( )*, *mq_timedsend( )*,
27295 *mq_unlink( )*, *msgctl( )*, *msgget( )*, *msgrcv( )*, *msgsnd( )*, the Base Definitions volume of
27296 IEEE Std 1003.1-200x, **<mqueue.h>**

27297 **CHANGE HISTORY**
27298 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

27299 **Issue 6**
27300 The *mq_open( )* function is marked as part of the Message Passing option.

27301 The [ENOSYS] error condition has been removed as stubs need not be provided if an
27302 implementation does not support the Message Passing option.

27303 The *mq_timedreceive( )* and *mq_timedsend( )* functions are added to the SEE ALSO section for
27304 alignment with IEEE Std 1003.1d-1999.

27305 The DESCRIPTION of O_EXCL is updated in response to IEEE PASC Interpretation 1003.1c #48.

27306 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/62 is applied, updating the description of
27307 the permission bits in the DESCRIPTION section. The change is made for consistency with the
27308 *shm_open( )* and *sem_open( )* functions.

27309 **Issue 7**
27310 Austin Group Interpretation 1003.1-2001 #077 is applied, clarifying the *name* argument and
27311 changing [ENAMETOOLONG] from a "shall fail" to a "may fail" error.

NAME

mq_receive, mq_timedreceive — receive a message from a message queue (**REALTIME**)

SYNOPSIS

MSG

```
#include <mqueue.h>

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
    unsigned *msg_prio);

#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.

The timeout shall be based on the CLOCK_REALTIME clock. 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 1003.1-200x, **<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**.

Issue 7

The *mq_timedreceive*( ) function is moved from the Timeouts option to the Base.

Functionality relating to the Timners option is moved to the Base.

NAME

mq_send, mq_timedsend — send a message to a message queue (**REALTIME**)

SYNOPSIS

MSG

```
#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.

The timeout shall be based on the CLOCK_REALTIME clock. 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 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.

27455 **RETURN VALUE**

27456 Upon successful completion, the *mq_send*( ) and *mq_timedsend*( ) functions shall return a value of
27457 zero. Otherwise, no message shall be enqueued, the functions shall return –1, and *errno* shall be
27458 set to indicate the error.

27459 **ERRORS**

27460 The *mq_send*( ) and *mq_timedsend*( ) functions shall fail if:

27461 [EAGAIN] The O_NONBLOCK flag is set in the message queue description associated
27462 with *mqdes*, and the specified message queue is full.

27463 [EBADF] The *mqdes* argument is not a valid message queue descriptor open for writing.

27464 [EINTR] A signal interrupted the call to *mq_send*( ) or *mq_timedsend*( ).

27465 [EINVAL] The value of *msg_prio* was outside the valid range.

27466 [EINVAL] The process or thread would have blocked, and the *abs_timeout* parameter
27467 specified a nanoseconds field value less than zero or greater than or equal to
27468 1 000 million.

27469 [EMSGSIZE] The specified message length, *msg_len*, exceeds the message size attribute of
27470 the message queue.

27471 [ETIMEDOUT] The O_NONBLOCK flag was not set when the message queue was opened,
27472 but the timeout expired before the message could be added to the queue.

27473 **EXAMPLES**

27474 None.

27475 **APPLICATION USAGE**

27476 The value of the symbol {MQ_PRIO_MAX} limits the number of priority levels supported by the
27477 application. Message priorities range from 0 to {MQ_PRIO_MAX}–1.

27478 **RATIONALE**

27479 None.

27480 **FUTURE DIRECTIONS**

27481 None.

27482 **SEE ALSO**

27483 *mq_open*( ), *mq_receive*( ), *mq_setattr*( ), *mq_timedreceive*( ), *time*( ), the Base Definitions volume of
27484 IEEE Std 1003.1-200x, **<mqueue.h>**, **<time.h>**

27485 **CHANGE HISTORY**

27486 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

27487 **Issue 6**

27488 The *mq_send*( ) function is marked as part of the Message Passing option.

27489 The [ENOSYS] error condition has been removed as stubs need not be provided if an
27490 implementation does not support the Message Passing option.

27491 The *mq_timedsend*( ) function is added for alignment with IEEE Std 1003.1d-1999.

27492 **Issue 7**

27493 The *mq_timedsend*( ) function is moved from the Timeouts option to the Base.

27494 Functionality relating to the Timers option is moved to the Base.

NAME

mq_setattr — set message queue attributes (**REALTIME**)

SYNOPSIS

MSG

```
#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.

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 1003.1-200x, **<mqueue.h>**

CHANGE HISTORY

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

Issue 6

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.

mq_timedreceive( )

27542 **NAME**

27543 mq_timedreceive — receive a message from a message queue (**ADVANCED REALTIME**)

27544 **SYNOPSIS**

27545 MSG
```
#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);
```

27550 **DESCRIPTION**

27551 Refer to *mq_receive( )*.

27552 **NAME**

27553 mq_timedsend — send a message to a message queue (**ADVANCED REALTIME**)

27554 **SYNOPSIS**

27555 MSG
```
#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);
```

27559 **DESCRIPTION**

27560 Refer to *mq_send( )*.

NAME

mq_unlink — remove a message queue (**REALTIME**)

SYNOPSIS

MSG
```
#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.

[ENOENT] The named message queue does not exist.

The *mq_unlink*( ) function may fail if:

[ENAMETOOLONG]
The length of the *name* argument exceeds {_POSIX_PATH_MAX} on systems that do not support the XSI option (XSI) or exceeds {_XOPEN_PATH_MAX} on XSI systems, or has a pathname component that is longer than {_POSIX_NAME_MAX} on systems that do not support the XSI option (XSI) or longer than {_XOPEN_NAME_MAX} on XSI systems. A call to *mq_unlink*( ) with a *name* argument that contains the same message queue name as was previously used in a successful *mq_open*( ) call shall not give an [ENAMETOOLONG] error.

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 1003.1-200x, **<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.

Issue 7

Austin Group Interpretation 1003.1-2001 #077 is applied, changing [ENAMETOOLONG] from a "shall fail" to a "may fail" error .

27615 **NAME**

27616 mrand48 — generate uniformly distributed pseudo-random signed long integers

27617 **SYNOPSIS**

27618 XSI
```
#include <stdlib.h>

long mrand48(void);
```

27620 **DESCRIPTION**

27621 Refer to *drand48( )*.

NAME

msgctl — XSI message control operations

SYNOPSIS

XSI
```
#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 1003.1-200x, 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 40).

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_RMD 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*.

[EPERM] 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 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 40), *mq_close( )*, *mq_getattr( )*, *mq_notify( )*, *mq_open( )*, *mq_receive( )*, *mq_send( )*, *mq_setattr( )*, *mq_unlink( )*, *msgget( )*, *msgrcv( )*, *msgsnd( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<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 1003.1-200x, 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 40).

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_ctime** 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).

[EEXIST] 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.

27729 **EXAMPLES**

27730 None.

27731 **APPLICATION USAGE**

27732 The POSIX Realtime Extension defines alternative interfaces for interprocess communication
27733 (IPC). Application developers who need to use IPC should design their applications so that
27734 modules using the IPC routines described in Section 2.7 can be easily modified to use the
27735 alternative interfaces.

27736 **RATIONALE**

27737 None.

27738 **FUTURE DIRECTIONS**

27739 None.

27740 **SEE ALSO**

27741 Section 2.7 (on page 39), Section 2.8 (on page 40), *mq_close( )*, *mq_getattr( )*, *mq_notify( )*,
27742 *mq_open( )*, *mq_receive( )*, *mq_send( )*, *mq_setattr( )*, *mq_unlink( )*, *msgctl( )*, *msgrcv( )*, *msgsnd( )*, the
27743 Base Definitions volume of IEEE Std 1003.1-200x, **<sys/msg.h>**

27744 **CHANGE HISTORY**

27745 First released in Issue 2. Derived from Issue 2 of the SVID.

27746 **Issue 5**

27747 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
27748 DIRECTIONS to a new APPLICATION USAGE section.

NAME

msgrcv — XSI message receive operation

SYNOPSIS

XSI
```
#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 1003.1-200x, 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 40).

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 *sigaction*( ).

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid*:

- **msg_qnum** shall be decremented by 1.
- **msg_lrpid** shall be set equal to the process ID of the calling process.
- **msg_rtime** shall be set equal to the current time.

RETURN VALUE

Upon successful completion, *msgrcv*( ) shall return a value equal to the number of bytes actually placed into the buffer *mtext*. Otherwise, no message shall be received, *msgrcv*( ) shall return (**ssize_t**)–1, and *errno* shall be set to indicate the error.

ERRORS

The *msgrcv*( ) function shall fail if:

[E2BIG] The value of *mtext* is greater than *msgsz* and (*msgflg* & 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 *msqid* is removed from the system.

[EINTR] The *msgrcv*( ) function was interrupted by a signal.

[EINVAL] *msqid* is not a valid message queue identifier.

[ENOMSG] The queue does not contain a message of the desired type and (*msgflg* & 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 *msgtyp* argument, 0). The queue is identified by the *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 *sizeof* operator.

```
#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);
```

27832 **APPLICATION USAGE**

27833 The POSIX Realtime Extension defines alternative interfaces for interprocess communication
27834 (IPC). Application developers who need to use IPC should design their applications so that
27835 modules using the IPC routines described in Section 2.7 can be easily modified to use the
27836 alternative interfaces.

27837 **RATIONALE**

27838 None.

27839 **FUTURE DIRECTIONS**

27840 None.

27841 **SEE ALSO**

27842 Section 2.7 (on page 39), Section 2.8 (on page 40), *mq_close( )*, *mq_getattr( )*, *mq_notify( )*,
27843 *mq_open( )*, *mq_receive( )*, *mq_send( )*, *mq_setattr( )*, *mq_unlink( )*, *msgctl( )*, *msgget( )*, *msgsnd( )*,
27844 *sigaction( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/msg.h>**

27845 **CHANGE HISTORY**

27846 First released in Issue 2. Derived from Issue 2 of the SVID.

27847 **Issue 5**

27848 The type of the return value is changed from **int** to **ssize_t**, and a warning is added to the
27849 DESCRIPTION about values of *msgsz* larger the {SSIZE_MAX}.

27850 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
27851 DIRECTIONS to the APPLICATION USAGE section.

27852 **Issue 6**

27853 The normative text is updated to avoid use of the term "must" for application requirements.

27854 **NAME**

27855 msgsnd — XSI message send operation

27856 **SYNOPSIS**

27857 XSI
```
#include <sys/msg.h>

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

27859 **DESCRIPTION**

27860 The *msgsnd*( ) function operates on XSI message queues (see the Base Definitions volume of
27861 IEEE Std 1003.1-200x, Section 3.224, Message Queue). It is unspecified whether this function
27862 interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on
27863 page 40).

27864 The *msgsnd*( ) function shall send a message to the queue associated with the message queue
27865 identifier specified by *msqid*.

27866 The application shall ensure that the argument *msgp* points to a user-defined buffer that contains
27867 first a field of type **long** specifying the type of the message, and then a data portion that holds
27868 the data bytes of the message. The structure below is an example of what this user-defined
27869 buffer might look like:

27870
```
struct mymsg {
    long   mtype;       /* Message type. */
    char   mtext[1];    /* Message text. */
}
```

27874 The structure member *mtype* is a non-zero positive type **long** that can be used by the receiving
27875 process for message selection.

27876 The structure member *mtext* is any text of length *msgsz* bytes. The argument *msgsz* can range
27877 from 0 to a system-imposed maximum.

27878 The argument *msgflg* specifies the action to be taken if one or more of the following is true:

27879 • The number of bytes already on the queue is equal to **msg_qbytes**; see **<sys/msg.h>**.

27880 • The total number of messages on all queues system-wide is equal to the system-imposed
27881 limit.

27882 These actions are as follows:

27883 • If (*msgflg* & IPC_NOWAIT) is non-zero, the message shall not be sent and the calling
27884 thread shall return immediately.

27885 • If (*msgflg* & IPC_NOWAIT) is 0, the calling thread shall suspend execution until one of the
27886 following occurs:

27887 — The condition responsible for the suspension no longer exists, in which case the
27888 message is sent.

27889 — The message queue identifier *msqid* is removed from the system; when this occurs,
27890 *errno* shall be set equal to [EIDRM] and –1 shall be returned.

27891 — The calling thread receives a signal that is to be caught; in this case the message is not
27892 sent and the calling thread resumes execution in the manner prescribed in *sigaction*( ).

27893 Upon successful completion, the following actions are taken with respect to the data structure
27894 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 *msqid* is removed from the system.

[EINTR] The *msgsnd*( ) function was interrupted by a signal.

[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.

```
#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 can be easily modified to use the alternative interfaces.

RATIONALE

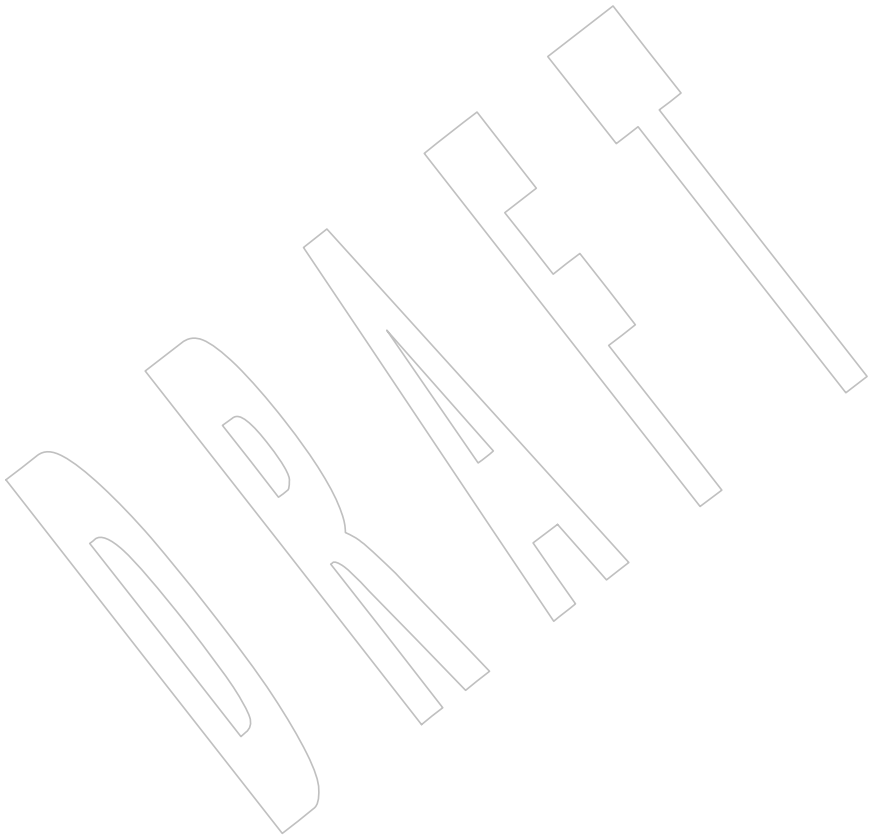
None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.7 (on page 39), Section 2.8 (on page 40), *mq_close( )*, *mq_getattr( )*, *mq_notify( )*, *mq_open( )*, *mq_receive( )*



NAME

msync — synchronize memory with physical storage

SYNOPSIS

SIO
```
#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 may 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
SHM | TYM 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:

Symbolic Constant	Description
MS_ASYNC	Perform asynchronous writes.
MS_SYNC	Perform synchronous writes.
MS_INVALIDATE	Invalidate cached data.

When MS_ASYNC 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 MS_ASYNC or MS_SYNC shall be specified, but not both.

When MS_INVALIDATE 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 MS_INVALIDATE is specified.

27993 [EINVAL] The value of *flags* is invalid.

27994 27995 27996 [ENOMEM] 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.

27997 The *msync*( ) function may fail if:

27998 [EINVAL] The value of *addr* is not a multiple of the page size as returned by *sysconf*( ).

27999 **EXAMPLES**

28000 None.

28001 **APPLICATION USAGE**

28002 28003 The *msync*( ) function is only supported if the Synchronized Input and Output option is supported, and thus need not be available on all implementations.

28004 28005 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.

28006 28007 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.

28008 **RATIONALE**

28009 28010 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.

28011 28012 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.

28013 **FUTURE DIRECTIONS**

28014 None.

28015 **SEE ALSO**

28016 *mmap*( ), *sysconf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/mman.h>**

28017 **CHANGE HISTORY**

28018 First released in Issue 4, Version 2.

28019 **Issue 5**

28020 Moved from X/OPEN UNIX extension to BASE.

28021 Aligned with *msync*( ) in the POSIX Realtime Extension as follows:

28022 - The DESCRIPTION is extensively reworded.
28023 - [EBUSY] and a new form of [EINVAL] are added as mandatory error conditions.

28024 **Issue 6**

28025 28026 The *msync*( ) function is marked as part of the Memory Mapped Files and Synchronized Input and Output options.

28027 The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

28028 - The [EBUSY] mandatory error condition is added.

28029 28030 The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

28031 28032 - The DESCRIPTION is updated to state that implementations require *addr* to be a multiple of the page size.
28033 - The second [EINVAL] error condition is made mandatory.

28034 28035 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding reference to typed memory objects.

Issue 7

SD5-XSH-ERN-110 is applied.

Austin Group Interpretation 1003.1-2001 #078 is applied, clarifying page alignment requirements.

The *msync*( ) function is moved from the Memory Mapped Files option.

28041 **NAME**

28042 munlock — unlock a range of process address space

28043 **SYNOPSIS**

28044 MLR
```
#include <sys/mman.h>

int munlock(const void *addr, size_t len);
```

28046 **DESCRIPTION**

28047 Refer to *mlock( )*.

28048 **NAME**

28049 munlockall — unlock the address space of a process

28050 **SYNOPSIS**

28051 ML
```
#include <sys/mman.h>

int munlockall(void);
```

28053 **DESCRIPTION**

28054 Refer to *mlockall( )*.

28055 **NAME**

28056 munmap — unmap pages of memory

28057 **SYNOPSIS**

28058
```
#include <sys/mman.h>
```

28059
```
int munmap(void *addr, size_t len);
```

28060 **DESCRIPTION**

28061 The *munmap*( ) function shall remove any mappings for those entire pages containing any part of
28062 the address space of the process starting at *addr* and continuing for *len* bytes. Further references
28063 to these pages shall result in the generation of a SIGSEGV signal to the process. If there are no
28064 mappings in the specified address range, then *munmap*( ) has no effect.

28065 The implementation may require that *addr* be a multiple of the page size as returned by
28066 *sysconf*( ).

28067 If a mapping to be removed was private, any modifications made in this address range shall be
28068 discarded.

28069 ML | MLR Any memory locks (see *mlock*( ) and *mlockall*( )) associated with this address range shall be
28070 removed, as if by an appropriate call to *munlock*( ).

28071 TYM If a mapping removed from a typed memory object causes the corresponding address range of
28072 the memory pool to be inaccessible by any process in the system except through allocatable
28073 mappings (that is, mappings of typed memory objects opened with the
28074 POSIX_TYPED_MEM_MAP_ALLOCATABLE flag), then that range of the memory pool shall
28075 become deallocated and may become available to satisfy future typed memory allocation
28076 requests.

28077 A mapping removed from a typed memory object opened with the
28078 POSIX_TYPED_MEM_MAP_ALLOCATABLE flag shall not affect in any way the availability of
28079 that typed memory for allocation.

28080 The behavior of this function is unspecified if the mapping was not established by a call to
28081 *mmap*( ).

28082 **RETURN VALUE**

28083 Upon successful completion, *munmap*( ) shall return 0; otherwise, it shall return –1 and set *errno*
28084 to indicate the error.

28085 **ERRORS**

28086 The *munmap*( ) function shall fail if:

28087 [EINVAL] Addresses in the range [*addr*,*addr*+*len*) are outside the valid range for the
28088 address space of a process.

28089 [EINVAL] The *len* argument is 0.

28090 The *munmap*( ) function may fail if:

28091 [EINVAL] The *addr* argument is not a multiple of the page size as returned by *sysconf*( ).

28092 **EXAMPLES**
28093 None.

28094 **APPLICATION USAGE**
28095 None.

28096 **RATIONALE**
28097 The *munmap*( ) function corresponds to SVR4, just as the *mmap*( ) function does.

28098 It is possible that an application has applied process memory locking to a region that contains
28099 shared memory. If this has occurred, the *munmap*( ) call ignores those locks and, if necessary,
28100 causes those locks to be removed.

28101 Most implementations require that *addr* is a multiple of the page size as returned by *sysconf*( ).

28102 **FUTURE DIRECTIONS**
28103 None.

28104 **SEE ALSO**
28105 *mlock*( ), *mlockall*( ), *mmap*( ), *posix_typed_mem_open*( ), *sysconf*( ), the Base Definitions volume of
28106 IEEE Std 1003.1-200x, **<signal.h>**, **<sys/mman.h>**

28107 **CHANGE HISTORY**
28108 First released in Issue 4, Version 2.

28109 **Issue 5**
28110 Moved from X/OPEN UNIX extension to BASE.

28111 Aligned with *munmap*( ) in the POSIX Realtime Extension as follows:

28112 - The DESCRIPTION is extensively reworded.
28113 - The SIGBUS error is no longer permitted to be generated.

28114 **Issue 6**
28115 The *munmap*( ) function is marked as part of the Memory Mapped Files and Shared Memory
28116 Objects option.

28117 The following new requirements on POSIX implementations derive from alignment with the
28118 Single UNIX Specification:

28119 - The DESCRIPTION is updated to state that implementations require *addr* to be a multiple
28120 of the page size.
28121 - The [EINVAL] error conditions are added.

28122 The following changes are made for alignment with IEEE Std 1003.1j-2000:

28123 - Semantics for typed memory objects are added to the DESCRIPTION.
28124 - The *posix_typed_mem_open*( ) function is added to the SEE ALSO section.

28125 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/36 is applied, changing the margin code
28126 in the SYNOPSIS from MF | SHM to MC3 (notation for MF | SHM | TYM).

28127 **Issue 7**
28128 Austin Group Interpretation 1003.1-2001 #078 is applied, clarifying page alignment
28129 requirements.

28130 The *munmap*( ) function is moved from the Memory Mapped Files option to the Base.

28131 **NAME**

28132 nan, nanf, nanl — return quiet NaN

28133 **SYNOPSIS**

28134
```
#include <math.h>

double nan(const char *tagp);
float nanf(const char *tagp);
long double nanl(const char *tagp);
```

28138 **DESCRIPTION**

28139 CX The functionality described on this reference page is aligned with the ISO C standard. Any
28140 conflict between the requirements described here and the ISO C standard is unintentional. This
28141 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

28142 The function call *nan*("*n-char-sequence*") shall be equivalent to:

28143
```
strtod("NAN(n-char-sequence)", (char **) NULL);
```

28144 The function call *nan*("") shall be equivalent to:

28145
```
strtod("NAN()", (char **) NULL)
```

28146 If *tagp* does not point to an *n*-**char** sequence or an empty string, the function call shall be
28147 equivalent to:

28148
```
strtod("NAN", (char **) NULL)
```

28149 Function calls to *nanf*( ) and *nanl*( ) are equivalent to the corresponding function calls to *strtof*( )
28150 and *strtold*( ).

28151 **RETURN VALUE**

28152 These functions shall return a quiet NaN, if available, with content indicated through *tagp*.

28153 If the implementation does not support quiet NaNs, these functions shall return zero.

28154 **ERRORS**

28155 No errors are defined.

28156 **EXAMPLES**

28157 None.

28158 **APPLICATION USAGE**

28159 None.

28160 **RATIONALE**

28161 None.

28162 **FUTURE DIRECTIONS**

28163 None.

28164 **SEE ALSO**

28165 *strtod*( ), *strtold*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

28166 **CHANGE HISTORY**

28167 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

28168 **NAME**

28169 nanosleep — high resolution sleep

28170 **SYNOPSIS**

28171 CX
```
#include <time.h>

int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);
```

28173 **DESCRIPTION**

28174 The *nanosleep*( ) function shall cause the current thread to be suspended from execution until
28175 either the time interval specified by the *rqtp* argument has elapsed or a signal is delivered to the
28176 calling thread, and its action is to invoke a signal-catching function or to terminate the process.
28177 The suspension time may be longer than requested because the argument value is rounded up to
28178 an integer multiple of the sleep resolution or because of the scheduling of other activity by the
28179 system. But, except for the case of being interrupted by a signal, the suspension time shall not be
28180 less than the time specified by *rqtp*, as measured by the system clock CLOCK_REALTIME.

28181 The use of the *nanosleep*( ) function has no effect on the action or blockage of any signal.

28182 **RETURN VALUE**

28183 If the *nanosleep*( ) function returns because the requested time has elapsed, its return value shall
28184 be zero.

28185 If the *nanosleep*( ) function returns because it has been interrupted by a signal, it shall return a
28186 value of –1 and set *errno* to indicate the interruption. If the *rmtp* argument is non-NULL, the
28187 **timespec** structure referenced by it is updated to contain the amount of time remaining in the
28188 interval (the requested time minus the time actually slept). The *rqtp* and *rmtp* arguments may
28189 point to the same object. If the *rmtp* argument is NULL, the remaining time is not returned.

28190 If *nanosleep*( ) fails, it shall return a value of –1 and set *errno* to indicate the error.

28191 **ERRORS**

28192 The *nanosleep*( ) function shall fail if:

28193 [EINTR] The *nanosleep*( ) function was interrupted by a signal.

28194 [EINVAL] The *rqtp* argument specified a nanosecond value less than zero or greater than
28195 or equal to 1 000 million.

28196 **EXAMPLES**

28197 None.

28198 **APPLICATION USAGE**

28199 None.

28200 **RATIONALE**

28201 It is common to suspend execution of a thread for an interval in order to poll the status of a non-
28202 interrupting function. A large number of actual needs can be met with a simple extension to
28203 *sleep*( ) that provides finer resolution.

28204 In the POSIX.1-1990 standard and SVR4, it is possible to implement such a routine, but the
28205 frequency of wakeup is limited by the resolution of the *alarm*( ) and *sleep*( ) functions. In 4.3 BSD,
28206 it is possible to write such a routine using no static storage and reserving no system facilities.
28207 Although it is possible to write a function with similar functionality to *sleep*( ) using the
28208 remainder of the *timer_**( ) functions, such a function requires the use of signals and the
28209 reservation of some signal number. This volume of IEEE Std 1003.1-200x requires that
28210 *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

clock_nanosleep( ), *sleep*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/37 is applied, updating the SEE ALSO section to include the *clock_nanosleep*( ) function.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/63 is applied, correcting text in the RATIONALE section.

Issue 7

SD5-XBD-ERN-33 is applied.

The *nanosleep*( ) function is moved from the Timers option to the Base.

NAME

nearbyint, nearbyintf, nearbyintl — floating-point rounding functions

SYNOPSIS

```
#include <math.h>

double nearbyint(double x);
float nearbyintf(float x);
long double nearbyintl(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 1003.1-200x defers to the ISO C standard.

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 *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.

MX 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.

XSI If the correct value would cause overflow, a range error shall occur and *nearbyint*( ), *nearbyintf*( ), and *nearbyintl*( ) shall return the value of the macro ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL (with the same sign as *x*), respectively.

ERRORS

These functions shall fail if:

XSI 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.

28270 **FUTURE DIRECTIONS**

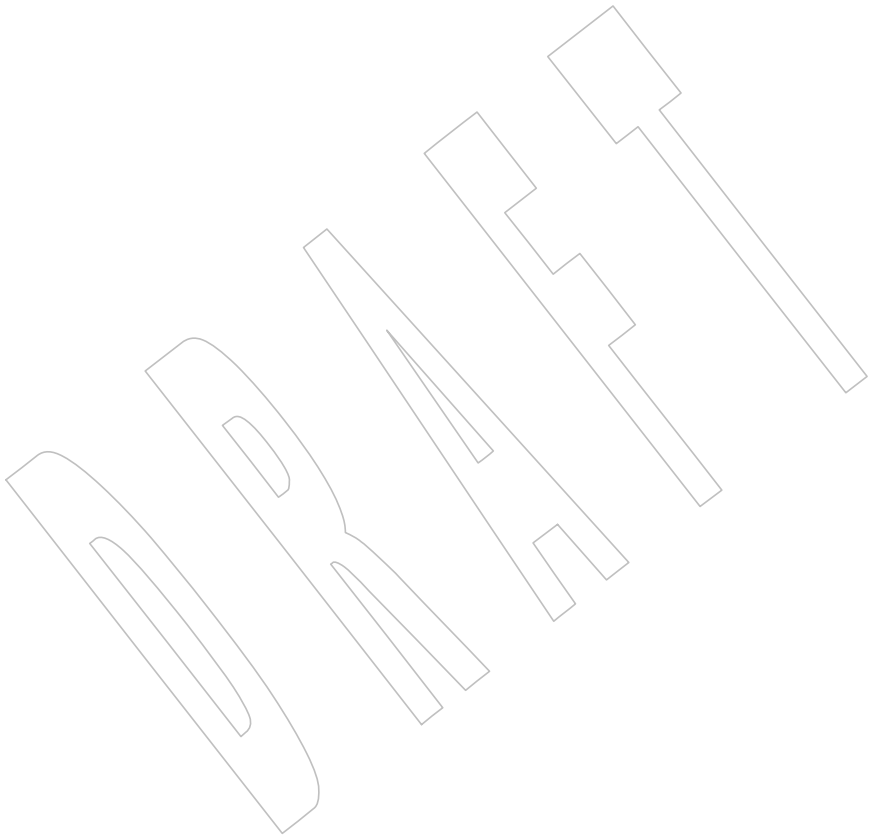
28271 None.

28272 **SEE ALSO**

28273 *feclearexcept*( ), *fetestexcept*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18,
28274 Treatment of Error Conditions for Mathematical Functions, **<math.h>**

28275 **CHANGE HISTORY**

28276 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.



NAME

newlocale — create or modify a locale object

SYNOPSIS

CX
```
#include <locale.h>

locale_t newlocale(int category_mask, const char *locale,
     locale_t base);
```

DESCRIPTION

The *newlocale*( ) function shall create a new locale object or modify an existing one. If the *base* argument is (**locale_t**)0, a new locale object shall be created. It is unspecified whether the locale object pointed to by *base* shall be modified or freed and a new locale object created.

The *category_mask* argument specifies the locale categories to be set or modified. Values for *category_mask* shall be constructed by a bitwise-inclusive OR of the symbolic constants *LC_CTYPE_MASK*, *LC_NUMERIC_MASK*, *LC_TIME_MASK*, *LC_COLLATE_MASK*, *LC_MONETARY_MASK*, and *LC_MESSAGES_MASK*, or any of the other implementation-defined *LC_*_MASK* values defined in **<locale.h>**.

For each category with the corresponding bit set in *category_mask* the data from the locale named by *locale* shall be used. In the case of modifying an existing locale object, the data from the locale named by *locale* shall replace the existing data within the locale object. If a completely new locale object is created, the data for all sections not requested by *category_mask* shall be taken from the default locale.

The following preset values of *locale* are defined for all settings of *category_mask*:

"POSIX" Specifies the minimal environment for C-language translation called the POSIX locale.

"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 1003.1-200x, Chapter 7, Locale and the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables.

If the *base* argument is not (**locale_t**)0 and the *newlocale*( ) function call succeeds, the contents of *base* are unspecified. Applications shall ensure that they stop using *base* as a locale object before calling *newlocale*( ). If the function call fails and the *base* argument is not (**locale_t**)0, the contents of *base* shall remain valid and unchanged.

The results are undefined if the *base* argument is the special locale object *LC_GLOBAL_LOCALE*.

RETURN VALUE

Upon successful completion, the *newlocale*( ) function shall return a handle which the caller may use on subsequent calls to *duplocale*( ), *freelocale*( ), and other functions taking a **locale_t** argument.

Upon failure, the *newlocale*( ) function shall return (**locale_t**)0 and set *errno* to indicate the error.

ERRORS

The *newlocale*( ) function shall fail if:

[ENOMEM] There is not enough memory available to create the locale object or load the locale data.

[EINVAL] The *category_mask* contains a bit that does not correspond to a valid category.

[ENOENT] For any of the categories in *category_mask,* the locale data is not available.

The *newlocale*( ) function may fail if:

[EINVAL] The *locale* argument is not a valid string pointer.

EXAMPLES

Constructing a Locale Object from Different Locales

The following example shows the construction of a locale where the *LC_CTYPE* category data comes from a locale *loc1* and the *LC_TIME* category data from a locale *tok2*:

```
#include <locale.h>
...
locale_t loc, new_loc;

/* Get the "loc1" data. */

loc = newlocale (LC_CTYPE_MASK, "loc1", NULL);
if (loc == (locale_t) 0)
    abort ();

/* Get the "loc2" data. */

new_loc = newlocale (LC_TIME_MASK, "loc2", loc);
if (new_loc != (locale_t) 0)
    /* We don t abort if this fails. In this case this
       simply used to unchanged locale object. */
    loc = new_loc;

...
```

Freeing up a Locale Object

The following example shows a code fragment to free a locale object created by *newlocale*( ):

```
#include <locale.h>
...

/* Every locale object allocated with newlocale() should be
 * freed using freelocale():
 */

locale_t loc;

/* Get the locale. */

loc = newlocale (LC_CTYPE_MASK | LC_TIME_MASK, "locname", NULL);

/* ... Use the locale object ... */
...

/* Free the locale object resources. */
freelocale (loc);
```

28356 **APPLICATION USAGE**

28357 Handles for locale objects created by the *newlocale*( ) function should be released by a
28358 corresponding call to *freelocale*( ).

28359 The special locale object *LC_GLOBAL_LOCALE* must not be passed for the *base* argument, even
28360 when returned by the *uselocale*( ) function.

28361 **RATIONALE**

28362 None.

28363 **FUTURE DIRECTIONS**

28364 None.

28365 **SEE ALSO**

28366 *duplocale*( ), *freelocale*( ), *uselocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
28367 **<locale.h>**

28368 **CHANGE HISTORY**

28369 First released in Issue 7.

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

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 1003.1-200x 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.

MX 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.

MX | Range Error | The correct value is subnormal or 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

feclearexcept( ), *fetestexcept( )*, the Base Definitions volume of IEEE Std 1003.1-200x, 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 *nextafter( )* function is no longer marked as an extension.

The *nextafterf( )*, *nextafterl( )*, *nexttoward( )*, *nexttowardf( )*, and *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.

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

NAME

nftw — walk a file tree

SYNOPSIS

XSI

```
#include <ftw.h>

int nftw(const char *path, int (*fn)(const char *,
    const struct stat *, int, struct FTW *), int fd_limit, 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, filled in as if *fstatat*( ), *stat*( ), or *lstat*( ) had been called to retrieve the information.
- 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 *fd_limit* 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 thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

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] All file descriptors available to the process are currently open.

[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, using a maximum of 5 file descriptors.

```
#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 fd_limit = 5;
int flags = FTW_CHDIR | FTW_DEPTH | FTW_MOUNT;
int ret;

ret = nftw(startpath, nftwfunc, fd_limit, flags);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fdopendir( ), *fstatat*( ), *readdir*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/64 is applied, changing the argument *depth* to *fd_limit* throughout and changing "to a maximum of 5 levels deep" to "using a maximum of 5 file descriptors" in the EXAMPLES section.

Issue 7

SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

SD6-XBD-ERN-61 is applied.

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 nice value of a process 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.

PS | TPS Calling the *nice*( ) function has no effect on the priority of processes or threads with policy SCHED_FIFO or SCHED_RR. The effect on processes or threads with other scheduling policies is implementation-defined.

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 nice value of the process 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.

28606 **RATIONALE**

28607 None.

28608 **FUTURE DIRECTIONS**

28609 None.

28610 **SEE ALSO**

28611 *exec*, *getpriority( )*, *setpriority( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<limits.h>**,

28612 **<unistd.h>**

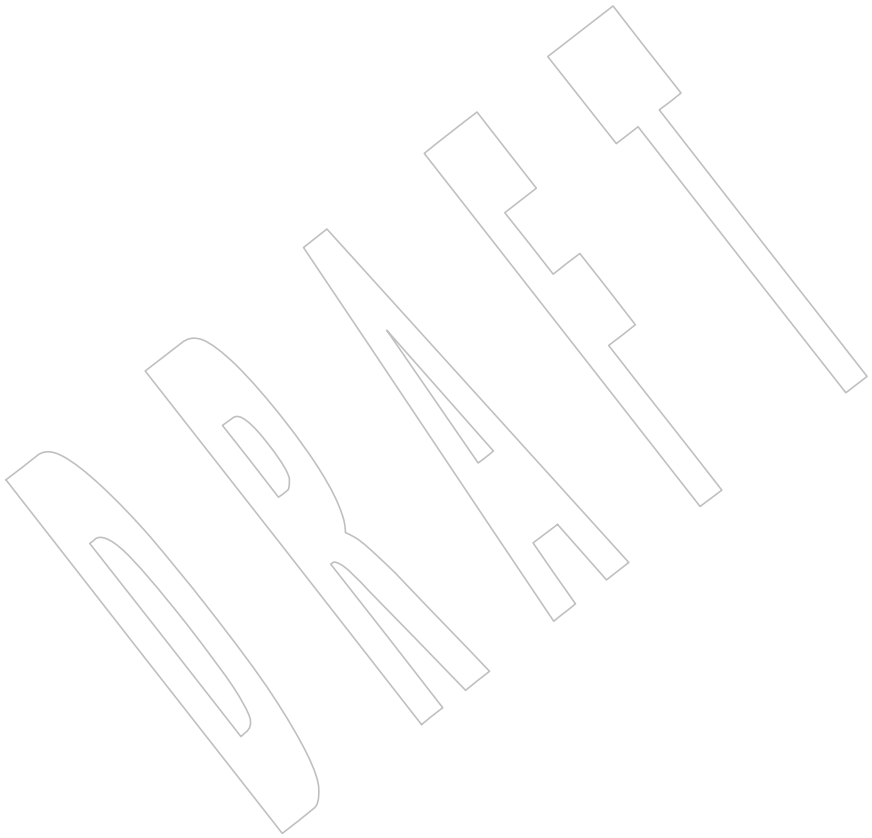
28613 **CHANGE HISTORY**

28614 First released in Issue 1. Derived from Issue 1 of the SVID.

28615 **Issue 5**

28616 A statement is added to the description indicating the effects of this function on the different

28617 scheduling policies and multi-threaded processes.



NAME

nl_langinfo, nl_langinfo_l — language information

SYNOPSIS

```
#include <langinfo.h>

char *nl_langinfo(nl_item item);
char *nl_langinfo_l(nl_item item, locale_t locale);
```

DESCRIPTION

The *nl_langinfo*( ) and *nl_langinfo_l*( ) functions shall return a pointer to a string containing information relevant to the particular language or cultural area defined in the locale of the process, or in the locale represented by *locale*, respectively (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.

```
nl_langinfo_l(ABDAY_1, loc)
```

would return a pointer to the string `"Dom"` if the identified language of the locale represented by *loc* was Portuguese, and `"Sun"` if the identified language of the locale represented by *loc* 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. Calls to *uselocale*( ) which change the category corresponding to the category of *item* may overwrite the array pointed to by the return value.

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

RETURN VALUE

In a locale where *langinfo* data is not defined, these functions shall return a pointer to the corresponding string in the POSIX locale. In all locales, these functions 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 to either function.

ERRORS

The *nl_langinfo_l*( ) function may fail if:

[EINVAL] *locale* is not a valid locale object handle.

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);
...
```

28659 **APPLICATION USAGE**

28660 The array pointed to by the return value should not be modified by the program, but may be
28661 modified by further calls to these functions.

28662 **RATIONALE**

28663 None.

28664 **FUTURE DIRECTIONS**

28665 None.

28666 **SEE ALSO**

28667 *setlocale*( ), *uselocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale,
28668 **<langinfo.h>**, **<locale.h>**, **<nl_types.h>**

28669 **CHANGE HISTORY**

28670 First released in Issue 2.

28671 **Issue 5**

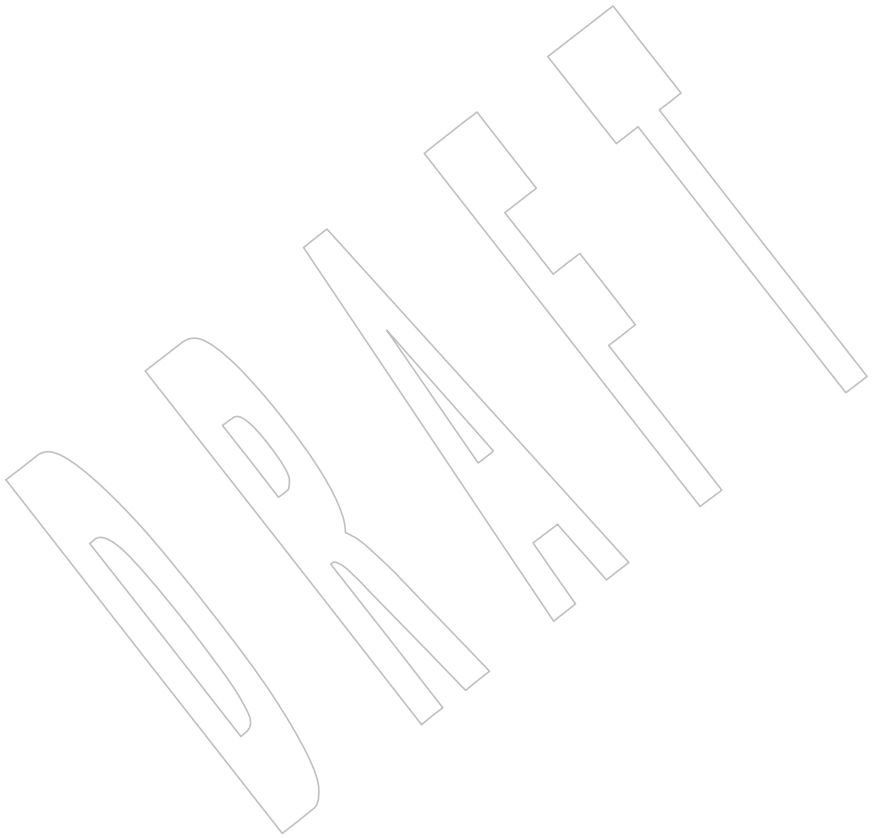
28672 The last paragraph of the DESCRIPTION is moved from the APPLICATION USAGE section.

28673 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

28674 **Issue 7**

28675 The *nl_langinfo*( ) function is moved from the XSI option to the Base.

28676 The *nl_langinfo_l*( ) function is added from The Open Group Technical Standard, 2006, Extended
28677 API Set Part 4.



28685 **NAME**

28686 ntohl, ntohs — convert values between host and network byte order

28687 **SYNOPSIS**

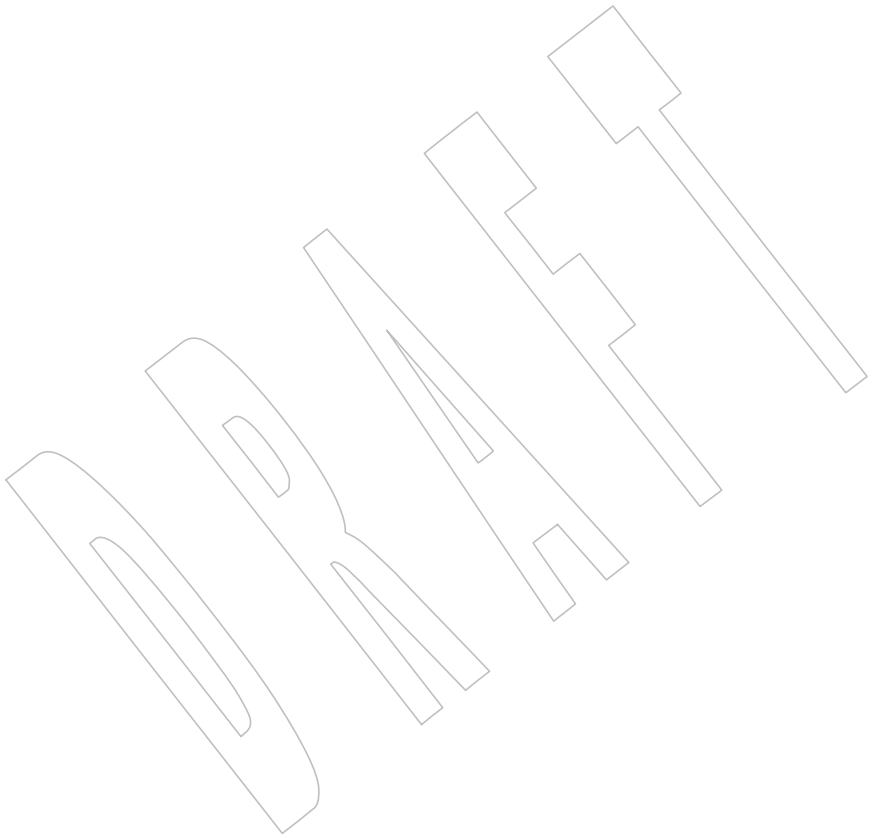
28688
```
#include <arpa/inet.h>
```

28689
```
uint32_t ntohl(uint32_t netlong);
```

28690
```
uint16_t ntohs(uint16_t netshort);
```

28691 **DESCRIPTION**

28692 Refer to *htonl( )*.



NAME

open, openat — open file relative to directory file descriptor

SYNOPSIS

OH
```
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *path, int oflag, ... );
int openat(int fd, 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 four values (file access modes) below in the value of *oflag*:

O_EXEC — Open for execute only (non-directory files). Use of this flag on directories is currently unspecified.

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 argument following the *oflag* 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 argument following the *oflag* 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.

SIO O_DIRECTORY — If *path* does not name a directory, fail and set *errno* to [ENOTDIR]. Write I/O operations on the file descriptor shall complete as defined by synchronized I/O data integrity completion.

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_NOFOLLOW — If *path* names a symbolic link, fail and set *errno* to [ELOOP]. 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.

SIO O_RSYNC — Read I/O operations on the file descriptor shall complete at the same level of integrity as specified by the O_DSYNC and O_SYNC flags. If both O_DSYNC and O_RSYNC are set in *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.

XSI | SIO O_SYNC — Write I/O operations on the file descriptor shall complete as defined by synchronized I/O file integrity completion.

XSI — The O_SYNC flag shall be supported for regular files, even if the Synchronized Input and Output option is not supported.

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 without either O_RDWR or O_WRONLY 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.

SIO If both the O_SYNC and O_DSYNC flags are set, the effect is as if only the O_SYNC flag was set.

OB XSR 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.

XSI 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.

The *openat*( ) function shall be equivalent to the *open*( ) function except in the case where *path* specifies a relative path. In this case the file to be opened is determined relative to the directory associated with the file descriptor *fd* instead of the current working directory. It is unspecified whether directory searches are permitted based on whether the file was opened with search permission or on the current permissions of the directory underlying the file descriptor.

The *oflag* parameter and the optional fourth parameter correspond exactly to the parameters of *open*( ).

If *openat*( ) is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior shall be identical to a call to *open*( ).

RETURN VALUE

Upon successful completion, these functions shall open the file and return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, these functions shall return –1 and set *errno* to indicate the error. If – is returned, no files shall be created or modified.

ERRORS

These functions 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.

[EINTR] A signal was caught during *open*( ).

SIO [EINVAL] The implementation does not support synchronized I/O for this file.

OB XSR [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, or O_NOFOLLOW was specified and the *path* argument names a symbolic link.

[EMFILE] All file descriptors available to the process are currently open.

[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.

OB XSR [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.

[ENOTDIR] A component of the path prefix is not a directory, or O_DIRECTORY was specified and the *path* argument does not name 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.

[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 either O_WRONLY, O_RDWR, O_CREAT (if the file does not exist), or O_TRUNC is set in the *oflag* argument.

The *openat( )* function shall fail if:

[EBADF] The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

These functions may fail if:

XSI [EAGAIN] The *path* argument names the slave side of a pseudo-terminal device that is locked.

[EINVAL] The value of the *oflag* argument is not valid.

[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}.

OB XSR [ENOMEM] The *path* argument names a STREAMS file and the system is unable to allocate resources.

[ETXTBSY] The file is a pure procedure (shared text) file that is being executed and *oflag* is O_WRONLY or O_RDWR.

The *openat*( ) function may fail if:

[ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

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.

```
#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.

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

#define LOCKFILE "/etc/ptmp"
...
int pfd; /* Integer for file descriptor returned by open() call. */
...
if ((pfd = open(LOCKFILE, O_WRONLY | O_CREAT | O_EXCL,
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
{
    fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n");
    exit(1);
}
...
```

Opening a File for Writing

The following example opens a file for writing, creating the file if it does not already exist. If the file does exist, the system truncates the file to zero bytes.

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

#define LOCKFILE "/etc/ptmp"
...
```

```
int pfd;
char filename[PATH_MAX+1];
...
if ((pfd = open(filename, O_WRONLY | O_CREAT | O_TRUNC,
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
{
    perror("Cannot open output file\n"); exit(1);
}
...
```

APPLICATION USAGE

None.

RATIONALE

Except as specified in this volume of IEEE Std 1003.1-200x, the flags allowed in *oflag* are not mutually-exclusive and any number of them may be used simultaneously.

Some implementations permit opening FIFOs with O_RDWR. Since FIFOs could be implemented in other ways, and since two file descriptors can be used to the same effect, this possibility is left as undefined.

See *getgroups( )* about the group of a newly created file.

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 1003.1-200x 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 1003.1-200x 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 1003.1-200x, 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
```

O_EXEC is specified as one of the four file access modes. On implementations where none of O_RDONLY, O_WRONLY, or O_RDWR is zero, applications may open a directory with O_EXEC OR'd in with one of the other three file access modes. On many historical implementations, this cannot be done since O_RDONLY has been defined to be zero.

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.

In addition, the *open*( ) function refuses to open non-directories if the O_DIRECTORY flag is set. This avoids race conditions whereby a user might compromise the system by substituting a hard link to a sensitive file (e.g., a device or a FIFO) while a privileged application is running, where opening a file even for read access might have undesirable side-effects.

In addition, the *open*( ) function does not follow symbolic links if the O_NOFOLLOW flag is set. This avoids race conditions whereby a user might compromise the system by substituting a symbolic link to a sensitive file (e.g., a device) while a privileged application is running, where opening a file even for read access might have undesirable side-effects.

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 nonexisting 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.

The purpose of the *openat*( ) function is to enable opening files in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to *open*( ), resulting in unspecified behavior. By opening a file descriptor for the target directory and using the *openat*( ) function it can be guaranteed that the opened file is located relative to the desired directory. Some implementations use the *openat*( ) function for other purposes as well. In some cases, if the *oflag* parameter has the O_XATTR bit set, the returned file descriptor provides access to extended attributes. This functionality is not standardized here.

FUTURE DIRECTIONS

The meaning of the O_EXEC flag on directories may be specified in a future version.

SEE ALSO

chmod( ), *close*( ), *creat*( ), *dirfd*( ), *dup*( ), *exec*, *fcntl*( ), *fdopendir*( ), *link*( ), *lseek*( ), *mkdtemp*( ), *mknod*( ), *read*( ), *symlink*( ), *umask*( ), *unlockpt*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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 normative text 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.

Issue 7

SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

This page is revised and the *openat*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 2.

Functionality relating to the XSI STREAMS option is marked obsolescent.

Austin Group Interpretation 1003.1-2001 #113 is applied.

29038 **NAME**

29039 open_memstream, open_wmemstream — open a dynamic memory buffer stream

29040 **SYNOPSIS**

29041 CX
```
#include <stdio.h>

FILE *open_memstream(char **bufp, size_t *sizep);

#include <wchar.h>

FILE *open_wmemstream(wchar_t **bufp, size_t *sizep);
```

29045 **DESCRIPTION**

29046 The *open_memstream*( ) and *open_wmemstream*( ) functions shall create an I/O stream associated
29047 with a dynamically allocated memory buffer. The stream shall be opened for writing and shall
29048 be seekable.

29049 The stream associated with a call to *open_memstream*( ) shall be byte-oriented.

29050 The stream associated with a call to *open_wmemstream*( ) shall be wide-oriented.

29051 The stream shall maintain a current position in the allocated buffer and a current buffer length.
29052 The position shall be initially set to zero (the start of the buffer). Each write to the stream shall
29053 start at the current position and move this position by the number of successfully written bytes
29054 for *open_memstream*( ) or the number of successfully written wide characters for
29055 *open_wmemstream*( ). The length shall be initially set to zero. If a write moves the position to a
29056 value larger than the current length, the current length shall be set to this position. In this case a
29057 null character for *open_memstream*( ) or a null wide character for *open_wmemstream*( ) shall be
29058 appended to the current buffer. For both functions the terminating null is not included in the
29059 calculation of the buffer length.

29060 After a successful *fflush*( ) or *fclose*( ), the pointer referenced by *bufp* shall contain the address of
29061 the buffer, and the variable pointed to by *sizep* shall contain the number of successfully written
29062 bytes for *open_memstream*( ) or the number of successfully written wide characters for
29063 *open_wmemstream*( ). The buffer shall be terminated by a null character for *open_memstream*( ) or
29064 a null wide character for *open_wmemstream*( ).

29065 After a successful *fflush*( ) the pointer referenced by *bufp* and the variable referenced by *sizep*
29066 remain valid only until the next write operation on the stream or a call to *fclose*( ).

29067 **RETURN VALUE**

29068 Upon successful completion, these functions shall return a pointer to the object controlling the
29069 stream. Otherwise, a null pointer shall be returned, and *errno* shall be set to indicate the error.

29070 **ERRORS**

29071 These functions may fail if:

29072 [EINVAL] *bufp* or *sizep* are NULL.

29073 [EMFILE] {FOPEN_MAX} streams are currently open in the calling process.

29074 [ENOMEM] Memory for the stream or the buffer could not be allocated.

EXAMPLES

```
#include <stdio.h>
int main (void)
{
    FILE *stream;
    char *buf;
    size_t len;

    stream = open_memstream(&buf, &len);

    if (stream == NULL)
        /* handle error */;

    fprintf(stream, "hello my world");
    fflush(stream);
    printf("buf=%s, len=%zu\n", buf, len);
    fseeko(stream, 0, SEEK_SET);
    fprintf(stream, "good-bye");
    fclose(stream);
    printf("buf=%s, len=%zu\n", buf, len);
    free(buf);
    return 0;
}
```

This program produces the following output:

```
buf=hello my world, len=14
buf=good-bye world, len=14
```

APPLICATION USAGE

The buffer created by these functions should be freed by the application after closing the stream, by means of a call to *free*( ).

RATIONALE

These functions are similar to *fmemopen*( ) except that the memory is always allocated dynamically by the function, and the stream is opened only for output.

FUTURE DIRECTIONS

None.

SEE ALSO

fclose( ), *fdopen*( ), *fflush*( ), *fmemopen*( ), *fopen*( ), *free*( ), *freopen*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 7.

open_wmemstream( )

29111 **NAME**

29112 open_wmemstream — open a dynamic memory buffer stream

29113 **SYNOPSIS**

29114 CX
```
#include <wchar.h>

FILE *open_wmemstream(wchar_t **bufp, size_t *sizep);
```

29116 **DESCRIPTION**

29117 Refer to *open_memstream( )*.

29118 **NAME**

29119 openat — open file relative to directory file descriptor

29120 **SYNOPSIS**

29121
```
#include <fcntl.h>
```

29122
```
int openat(int fd, const char *path, int oflag, ...);
```

29123 **DESCRIPTION**

29124 Refer to *open( )*.

29125 **NAME**

29126 opendir — open directory associated with file descriptor

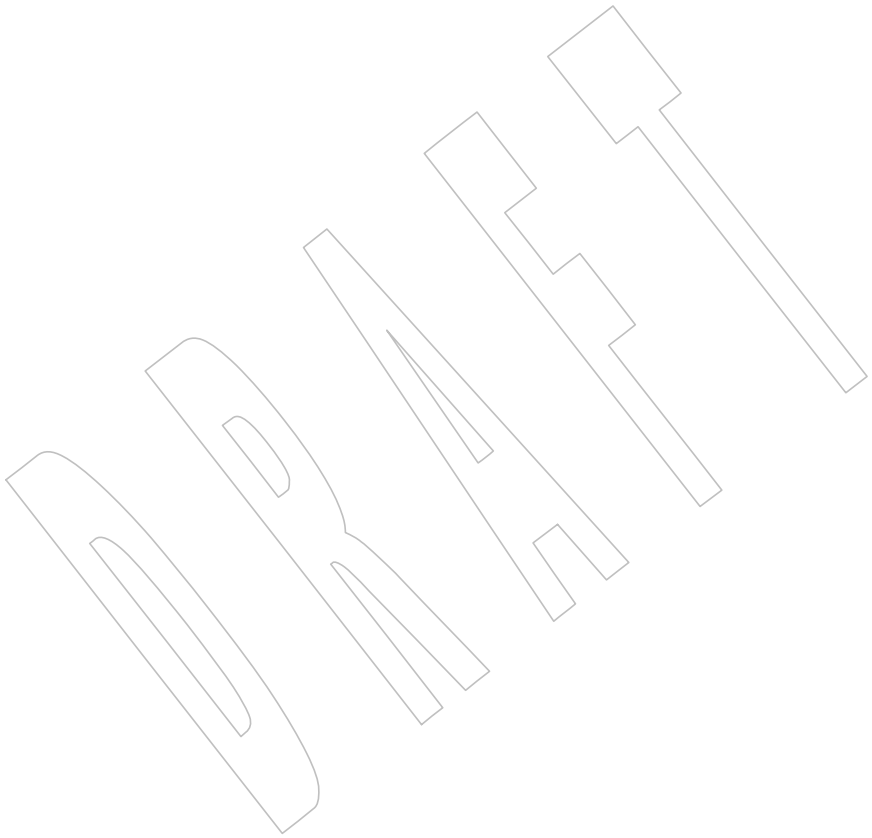
29127 **SYNOPSIS**

29128
```
#include <dirent.h>
```

29129
```
DIR *opendir(const char *dirname);
```

29130 **DESCRIPTION**

29131 Refer to *fdopendir( )*.



29132 **NAME**

29133 openlog — open a connection to the logging facility

29134 **SYNOPSIS**

29135 XSI
```
#include <syslog.h>

void openlog(const char *ident, int logopt, int facility);
```

29137 **DESCRIPTION**

29138 Refer to *closelog( )*.

29139 **NAME**

29140 optarg, opterr, optind, optopt — options parsing variables

29141 **SYNOPSIS**

29142
```
#include <unistd.h>
```

29143
```
extern char *optarg;
```
29144
```
extern int opterr, optind, optopt;
```

29145 **DESCRIPTION**

29146 Refer to *getopt*( ).

29147 **NAME**

29148 pathconf — get configurable pathname variables

29149 **SYNOPSIS**

29150
```
#include <unistd.h>
```

29151
```
long pathconf(const char *path, int name);
```

29152 **DESCRIPTION**

29153 Refer to *fpathconf( )*.

29154 **NAME**

29155 pause — suspend the thread until a signal is received

29156 **SYNOPSIS**

29157
```
#include <unistd.h>
```

29158
```
int pause(void);
```

29159 **DESCRIPTION**

29160 The *pause*( ) function shall suspend the calling thread until delivery of a signal whose action is
29161 either to execute a signal-catching function or to terminate the process.

29162 If the action is to terminate the process, *pause*( ) shall not return.

29163 If the action is to execute a signal-catching function, *pause*( ) shall return after the signal-catching
29164 function returns.

29165 **RETURN VALUE**

29166 Since *pause*( ) suspends thread execution indefinitely unless interrupted by a signal, there is no
29167 successful completion return value. A value of –1 shall be returned and *errno* set to indicate the
29168 error.

29169 **ERRORS**

29170 The *pause*( ) function shall fail if:

29171 [EINTR] A signal is caught by the calling process and control is returned from the
29172 signal-catching function.

29173 **EXAMPLES**

29174 None.

29175 **APPLICATION USAGE**

29176 Many common uses of *pause*( ) have timing windows. The scenario involves checking a
29177 condition related to a signal and, if the signal has not occurred, calling *pause*( ). When the signal
29178 occurs between the check and the call to *pause*( ), the process often blocks indefinitely. The
29179 *sigprocmask*( ) and *sigsuspend*( ) functions can be used to avoid this type of problem.

29180 **RATIONALE**

29181 None.

29182 **FUTURE DIRECTIONS**

29183 None.

29184 **SEE ALSO**

29185 *sigsuspend*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

29186 **CHANGE HISTORY**

29187 First released in Issue 1. Derived from Issue 1 of the SVID.

29188 **Issue 5**

29189 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

29190 **Issue 6**

29191 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 1003.1-200x 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 1003.1-200x 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 1003.1-200x, such implementations are not conforming. Also, some historical implementations return [EINTR] 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:

```
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 1003.1-200x, *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 1003.1-200x.

```
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 1003.1-200x, **<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

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 1003.1-200x 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*.

CX 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.

```
#include <stdio.h>
#include <stdlib.h>
...
char *bufptr;
size_t szbuf;
...
if ((bufptr = malloc(szbuf)) == NULL) {
    perror("malloc"); exit(2);
}
...
```


29313 **APPLICATION USAGE**

29314 None.

29315 **RATIONALE**

29316 None.

29317 **FUTURE DIRECTIONS**

29318 None.

29319 **SEE ALSO**

29320 *psiginfo*( ), *strerror*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

29321 **CHANGE HISTORY**

29322 First released in Issue 1. Derived from Issue 1 of the SVID.

29323 **Issue 5**

29324 A paragraph is added to the DESCRIPTION indicating that *perror*( ) does not change the
29325 orientation of the standard error stream.

29326 **Issue 6**

29327 Extensions beyond the ISO C standard are marked.

29328 **NAME**

29329 pipe — create an interprocess channel

29330 **SYNOPSIS**

29331 ```
#include <unistd.h>
```

29332 ```
int pipe(int fildes[2]);
```

29333 **DESCRIPTION**

29334 The *pipe*( ) function shall create a pipe and place two file descriptors, one each into the
29335 arguments *fildes*[0] and *fildes*[1], that refer to the open file descriptions for the read and write
29336 ends of the pipe. Their integer values shall be the two lowest available at the time of the *pipe*( )
29337 call. The O_NONBLOCK and FD_CLOEXEC flags shall be clear on both file descriptors. (The
29338 *fcntl*( ) function can be used to set both these flags.)

29339 Data can be written to the file descriptor *fildes*[1] and read from the file descriptor *fildes*[0]. A
29340 read on the file descriptor *fildes*[0] shall access data written to the file descriptor *fildes*[1] on a
29341 first-in-first-out basis. It is unspecified whether *fildes*[0] is also open for writing and whether
29342 *fildes*[1] is also open for reading.

29343 A process has the pipe open for reading (correspondingly writing) if it has a file descriptor open
29344 that refers to the read end, *fildes*[0] (write end, *fildes*[1]).

29345 The pipe's user ID shall be set to the effective user ID of the calling process.

29346 The pipe's group ID shall be set to the effective group ID of the calling process.

29347 Upon successful completion, *pipe*( ) shall mark for update the *st_atime*, *st_ctime*, and *st_mtime*
29348 fields of the pipe.

29349 **RETURN VALUE**

29350 Upon successful completion, 0 shall be returned; otherwise, –1 shall be returned and *errno* set to
29351 indicate the error.

29352 **ERRORS**

29353 The *pipe*( ) function shall fail if:

29354 [EMFILE] All, or all but one, of the file descriptors available to the process are currently
29355 open.

29356 [ENFILE] The number of simultaneously open files in the system would exceed a
29357 system-imposed limit.

29358 **EXAMPLES**

29359 **Using a Pipe to Pass Data Between a Parent Process and a Child Process**

29360 The following example demonstrates the use of a pipe to transfer data between a parent process
29361 and a child process. Error handling is excluded, but otherwise this code demonstrates good
29362 practice when using pipes: after the *fork*( ) the two processes close the unused ends of the pipe
29363 before they commence transferring data.

29364 ```
#include <stdlib.h>
```
29365 ```
#include <unistd.h>
```
29366 ```
...
```

29367 ```
int fildes[2];
```
29368 ```
const int BSIZE = 100;
```
29369 ```
char buf[BSIZE];
```
29370 ```
ssize_t nbytes;
```

  
```

```
int status;

status = pipe(fildes);
if (status == –1 ) {
    /* an error occurred */
    ...
}

switch (fork()) {
case –1: /* Handle error */
    break;

case 0:  /* Child - reads from pipe */
    close(fildes[1]);                       /* Write end is unused */
    nbytes = read(fildes[0], buf, BSIZE);   /* Get data from pipe */
    /* At this point, a further read would see end of file ... */
    close(fildes[0]);                       /* Finished with pipe */
    exit(EXIT_SUCCESS);

default:  /* Parent - writes to pipe */
    close(fildes[0]);                       /* Read end is unused */
    write(fildes[1], "Hello world\n", 12);  /* Write data on pipe */
    close(fildes[1]);                       /* Child will see EOF */
    exit(EXIT_SUCCESS);
}
```

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 1003.1-200x, **<fcntl.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 certain dispositions of *fildes*[0] and *fildes*[1] are unspecified.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/65 is applied, adding the example to the EXAMPLES section.

Issue 7

SD5-XSH-ERN-156 is applied, updating the DESCRIPTION to state the setting of the pipe's user ID and group ID.

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.

OB XSR — 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.

OB XSR — 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.

OB XSR — 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.

OB XSR — For STREAMS, this flag is set in *revents* even if the message is of zero length.

POLLOUT — Normal data may be written without blocking.

OB XSR — For STREAMS, data on priority band 0 may be written without blocking.

POLLWRNORM — Equivalent to POLLOUT.

POLLWRBAND — Priority data may be written.

OB XSR — 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.

29455 POLLNVAL The specified *fd* value is invalid. This flag is only valid in the *revents* member;
29456 it shall ignored in the *events* member.

29457 The significance and semantics of normal, priority, and high-priority data are file and device-
29458 specific.

29459 If the value of *fd* is less than 0, *events* shall be ignored, and *revents* shall be set to 0 in that entry on
29460 return from *poll*( ).

29461 In each **pollfd** structure, *poll*( ) shall clear the *revents* member, except that where the application
29462 requested a report on a condition by setting one of the bits of *events* listed above, *poll*( ) shall set
29463 the corresponding bit in *revents* if the requested condition is true. In addition, *poll*( ) shall set the
29464 POLLHUP, POLLERR, and POLLNVAL flag in *revents* if the condition is true, even if the
29465 application did not set the corresponding bit in *events*.

29466 If none of the defined events have occurred on any selected file descriptor, *poll*( ) shall wait at
29467 least *timeout* milliseconds for an event to occur on any of the selected file descriptors. If the value
29468 of *timeout* is 0, *poll*( ) shall return immediately. If the value of *timeout* is –1, *poll*( ) shall block until
29469 a requested event occurs or until the call is interrupted.

29470 Implementations may place limitations on the granularity of timeout intervals. If the requested
29471 timeout interval requires a finer granularity than the implementation supports, the actual
29472 timeout interval shall be rounded up to the next supported value.

29473 The *poll*( ) function shall not be affected by the O_NONBLOCK flag.

29474 The *poll*( ) function shall support regular files, terminal and pseudo-terminal devices, FIFOs,
29475 OB XSR pipes, sockets and STREAMS-based files. The behavior of *poll*( ) on elements of *fds* that refer to
29476 other types of file is unspecified.

29477 Regular files shall always poll TRUE for reading and writing.

29478 A file descriptor for a socket that is listening for connections shall indicate that it is ready for
29479 reading, once connections are available. A file descriptor for a socket that is connecting
29480 asynchronously shall indicate that it is ready for writing, once a connection has been established.

29481 **RETURN VALUE**

29482 Upon successful completion, *poll*( ) shall return a non-negative value. A positive value indicates
29483 the total number of file descriptors that have been selected (that is, file descriptors for which the
29484 *revents* member is non-zero). A value of 0 indicates that the call timed out and no file descriptors
29485 have been selected. Upon failure, *poll*( ) shall return –1 and set *errno* to indicate the error.

29486 **ERRORS**

29487 The *poll*( ) function shall fail if:

29488 [EAGAIN] The allocation of internal data structures failed but a subsequent request may
29489 succeed.

29490 [EINTR] A signal was caught during *poll*( ).

29491 OB XSR [EINVAL] The *nfds* argument is greater than {OPEN_MAX}, or one of the *fd* members
29492 refers to a STREAM or multiplexer that is linked (directly or indirectly)
29493 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 polls 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 1003.1-200x.
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. */
...
        }
        if (fds[i].revents & POLLOUT) {
        /* 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 1003.1-200x, **<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.

Functionality relating to the XSI STREAMS Option Group is marked.

The Open Group Corrigendum U055/3 is applied, updating the DESCRIPTION of POLLWRBAND.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/66 is applied, correcting the spacing in the EXAMPLES section.

Issue 7

The *poll( )* function is moved from the XSI option to the Base.

Functionality relating to the XSI STREAMS option is marked obsolescent.

NAME

popen — initiate pipe streams to or from a process

SYNOPSIS

CX
```
#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 unspecified.

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

Using popen( ) to Obtain a List of Files from the ls Utility

The following example demonstrates the use of *popen*( ) and *pclose*( ) to execute the command *ls** in order to obtain a list of files in the current directory:

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

FILE *fp;
int status;
char path[PATH_MAX];

fp = popen("ls *", "r");
if (fp == NULL)
    /* Handle error */;

while (fgets(path, PATH_MAX, fp) != NULL)
    printf("%s", path);

status = pclose(fp);
if (status == -1) {
    /* Error reported by pclose() */
    ...
} else {
    /* Use macros described under wait() to inspect 'status' in order
       to determine success/failure of command executed by popen() */
    ...
}
```

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 1003.1-200x is present, use the function call:

```
sysconf(_SC_2_VERSION)
```

(See *sysconf*( )).

29651 **RATIONALE**

29652 The *popen*( ) function should not be used by programs that have set user (or group) ID privileges.
29653 The *fork*( ) and *exec* family of functions (except *execlp*( ) and *execvp*( )), should be used instead.
29654 This prevents any unforeseen manipulation of the environment of the user that could cause
29655 execution of commands not anticipated by the calling program.

29656 If the original and *popen*( )ed processes both intend to read or write or read and write a common
29657 file, and either will be using FILE-type C functions (*fread*( ), *fwrite*( ), and so on), the rules for
29658 sharing file handles must be observed (see Section 2.5.1 (on page 35)).

29659 **FUTURE DIRECTIONS**

29660 None.

29661 **SEE ALSO**

29662 *pclose*( ), *pipe*( ), *sysconf*( ), *system*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
29663 **<stdio.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x, *sh*

29664 **CHANGE HISTORY**

29665 First released in Issue 1. Derived from Issue 1 of the SVID.

29666 **Issue 5**

29667 A statement is added to the DESCRIPTION indicating that pipe streams are byte-oriented.

29668 **Issue 6**

29669 The following new requirements on POSIX implementations derive from alignment with the
29670 Single UNIX Specification:

29671 - The optional [EMFILE] error condition is added.

29672 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/67 is applied, adding the example to the
29673 EXAMPLES section.

29674 **Issue 7**

29675 Austin Group Interpretation 1003.1-2001 #029 is applied, clarifying the values for *mode* in the
29676 DESCRIPTION.

29677 **NAME**

29678 posix_fadvise — file advisory information (**ADVANCED REALTIME**)

29679 **SYNOPSIS**

29680 ADV
```
#include <fcntl.h>

int posix_fadvise(int fd, off_t offset, off_t len, int advice);
```

29682 **DESCRIPTION**

29683 The *posix_fadvise*( ) function shall advise the implementation on the expected behavior of the
29684 application with respect to the data in the file associated with the open file descriptor, *fd*, starting
29685 at *offset* and continuing for *len* bytes. The specified range need not currently exist in the file. If *len*
29686 is zero, all data following *offset* is specified. The implementation may use this information to
29687 optimize handling of the specified data. The *posix_fadvise*( ) function shall have no effect on the
29688 semantics of other operations on the specified data, although it may affect the performance of
29689 other operations.

29690 The advice to be applied to the data is specified by the *advice* parameter and may be one of the
29691 following values:

29692 POSIX_FADV_NORMAL
29693 Specifies that the application has no advice to give on its behavior with respect to the
29694 specified data. It is the default characteristic if no advice is given for an open file.

29695 POSIX_FADV_SEQUENTIAL
29696 Specifies that the application expects to access the specified data sequentially from lower
29697 offsets to higher offsets.

29698 POSIX_FADV_RANDOM
29699 Specifies that the application expects to access the specified data in a random order.

29700 POSIX_FADV_WILLNEED
29701 Specifies that the application expects to access the specified data in the near future.

29702 POSIX_FADV_DONTNEED
29703 Specifies that the application expects that it will not access the specified data in the near
29704 future.

29705 POSIX_FADV_NOREUSE
29706 Specifies that the application expects to access the specified data once and then not reuse it
29707 thereafter.

29708 These values are defined in **<fcntl.h>**.

29709 **RETURN VALUE**

29710 Upon successful completion, *posix_fadvise*( ) shall return zero; otherwise, an error number shall
29711 be returned to indicate the error.

29712 **ERRORS**

29713 The *posix_fadvise*( ) function shall fail if:

29714 [EBADF] The *fd* argument is not a valid file descriptor.

29715 [EINVAL] The value of *advice* is invalid, or the value of *len* is less than zero.

29716 [ESPIPE] The *fd* argument is associated with a pipe or FIFO.

29717 **EXAMPLES**

29718 None.

29719 **APPLICATION USAGE**

29720 The *posix_fadvise*( ) function is part of the Advisory Information option and need not be
29721 provided on all implementations.

29722 **RATIONALE**

29723 None.

29724 **FUTURE DIRECTIONS**

29725 None.

29726 **SEE ALSO**

29727 *posix_madvise*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fcntl.h>**

29728 **CHANGE HISTORY**

29729 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

29730 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

29731 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/68 is applied, changing the function
29732 prototype in the SYNOPSIS section. The previous prototype was not large file-aware, and the
29733 standard developers felt it acceptable to make this change before implementations of this
29734 function become widespread.

29735 **Issue 7**

29736 Austin Group Interpretation 1003.1-2001 #024 is applied, changing the definition of the
29737 [EINVAL] error.

NAME

posix_fallocate — file space control (**ADVANCED REALTIME**)

SYNOPSIS

ADV

```
#include <fcntl.h>

int posix_fallocate(int fd, off_t offset, off_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 is less than or equal to zero, or the *offset* argument is less than zero, or the underlying file system does not support this operation.

[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.

29775 **RATIONALE**

29776 None.

29777 **FUTURE DIRECTIONS**

29778 None.

29779 **SEE ALSO**

29780 *creat( )*, *ftruncate( )*, *open( )*, *unlink( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<fcntl.h>**

29782 **CHANGE HISTORY**

29783 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

29784 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

29785 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/69 is applied, changing the function
29786 prototype in the SYNOPSIS section. The previous prototype was not large file-aware, and the
29787 standard developers felt it acceptable to make this change before implementations of this
29788 function become widespread.

29789 **Issue 7**

29790 Austin Group Interpretations 1003.1-2001 #022 and #024 are applied, changing the definition of
29791 the [EINVAL] error.

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 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:

[EINVAL] The value of *addr* is not a multiple of the value returned by *sysconf*( ) when the name value _SC_PAGESIZE is used.

[EINVAL] The value of *len* is zero.

EXAMPLES

None.

APPLICATION USAGE

The *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

mmap( ), *posix_fadvise*( ), *sysconf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/mman.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

IEEE PASC Interpretation 1003.1 #102 is applied.

29850 **NAME**

29851 posix_mem_offset — find offset and length of a mapped typed memory block (**ADVANCED**
29852 **REALTIME**)

29853 **SYNOPSIS**

29854 TYM
```
#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);
```

29858 **DESCRIPTION**

29859 The *posix_mem_offset*( ) function shall return in the variable pointed to by *off* a value that
29860 identifies the offset (or location), within a memory object, of the memory block currently
29861 mapped at *addr*. The function shall return in the variable pointed to by *fildes*, the descriptor used
29862 (via *mmap*( )) to establish the mapping which contains *addr*. If that descriptor was closed since
29863 the mapping was established, the returned value of *fildes* shall be –1. The *len* argument specifies
29864 the length of the block of the memory object the user wishes the offset for; upon return, the
29865 value pointed to by *contig_len* shall equal either *len*, or the length of the largest contiguous block
29866 of the memory object that is currently mapped to the calling process starting at *addr*, whichever
29867 is smaller.

29868 If the memory object mapped at *addr* is a typed memory object, then if the *off* and *contig_len*
29869 values obtained by calling *posix_mem_offset*( ) are used in a call to *mmap*( ) with a file descriptor
29870 that refers to the same memory pool as *fildes* (either through the same port or through a different
29871 port), and that was opened with neither the POSIX_TYPED_MEM_ALLOCATE nor the
29872 POSIX_TYPED_MEM_ALLOCATE_CONTIG flag, the typed memory area that is mapped shall
29873 be exactly the same area that was mapped at *addr* in the address space of the process that called
29874 *posix_mem_offset*( ).

29875 If the memory object specified by *fildes* is not a typed memory object, then the behavior of this
29876 function is implementation-defined.

29877 **RETURN VALUE**

29878 Upon successful completion, the *posix_mem_offset*( ) function shall return zero; otherwise, the
29879 corresponding error status value shall be returned.

29880 **ERRORS**

29881 The *posix_mem_offset*( ) function shall fail if:

29882 [EACCES] The process has not mapped a memory object supported by this function at
29883 the given address *addr*.

29884 This function shall not return an error code of [EINTR].

29885 **EXAMPLES**

29886 None.

29887 **APPLICATION USAGE**

29888 None.

29889 **RATIONALE**

29890 None.

29891 **FUTURE DIRECTIONS**

29892 None.

29893 **SEE ALSO**

29894 *mmap( )*, *posix_typed_mem_open( )*, the Base Definitions volume of IEEE Std 1003.1-200x,
29895 **<sys/mman.h>**

29896 **CHANGE HISTORY**

29897 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

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 power of two multiple of *sizeof* (**void** *). Upon successful completion, the value pointed to by *memptr* shall be a multiple of *alignment*.

CX 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 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

Issue 7

Austin Group Interpretation 1003.1-2001 #058 is applied, clarifying the value of the *alignment* argument in the DESCRIPTION.

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 device and return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *posix_openpt*( ) function shall fail if:

[EMFILE] All file descriptors available to the process are currently open.

[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.

OB 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
        || unlockpt (masterfd) == -1
        || (slavedevice = ptsname (masterfd)) == NULL)
        return -1;

    printf("slave device is: %s\n", slavedevice);

    slavefd = open(slavedevice, 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:

```
mfdp = open("/dev/ptmx", oflag );
if (mfdp < 0)
    return -1;
```

FUTURE DIRECTIONS

None.

SEE ALSO

grantpt( ), *open*( ), *ptsname*( ), *unlockpt*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fcntl.h>**

CHANGE HISTORY

First released in Issue 6.

Issue 7

SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

SD5-XSH-ERN-51 is applied, correcting an error in the EXAMPLES section.

30006 **NAME**

30007 posix_spawn, posix_spawnp — spawn a process (**ADVANCED REALTIME**)

30008 **SYNOPSIS**

30009 SPN
```
#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]);
```

30018 **DESCRIPTION**

30019 The *posix_spawn*( ) and *posix_spawnp*( ) functions shall create a new process (child process) from
30020 the specified process image. The new process image shall be constructed from a regular
30021 executable file called the new process image file.

30022 When a C program is executed as the result of this call, it shall be entered as a C-language
30023 function call as follows:

30024 `int main(int argc, char *argv[]);`

30025 where *argc* is the argument count and *argv* is an array of character pointers to the arguments
30026 themselves. In addition, the following variable:

30027 `extern char **environ;`

30028 shall be initialized as a pointer to an array of character pointers to the environment strings.

30029 The argument *argv* is an array of character pointers to null-terminated strings. The last member
30030 of this array shall be a null pointer and is not counted in *argc*. These strings constitute the
30031 argument list available to the new process image. The value in *argv*[0] should point to a filename
30032 that is associated with the process image being started by the *posix_spawn*( ) or *posix_spawnp*( )
30033 function.

30034 The argument *envp* is an array of character pointers to null-terminated strings. These strings
30035 constitute the environment for the new process image. The environment array is terminated by a
30036 null pointer.

30037 The number of bytes available for the combined argument and environment lists of the child
30038 process is {ARG_MAX}. The implementation shall specify in the system documentation (see the
30039 Base Definitions volume of IEEE Std 1003.1-200x, Chapter 2, Conformance) whether any list
30040 overhead, such as length words, null terminators, pointers, or alignment bytes, is included in
30041 this total.

30042 The *path* argument to *posix_spawn*( ) is a pathname that identifies the new process image file to
30043 execute.

30044 The *file* parameter to *posix_spawnp*( ) shall be used to construct a pathname that identifies the
30045 new process image file. If the *file* parameter contains a slash character, the *file* parameter shall be
30046 used as the pathname for the new process image file. Otherwise, the path prefix for this file shall
30047 be obtained by a search of the directories passed as the environment variable *PATH* (see the Base
30048 Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables). If this
30049 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.

PS 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 effective user ID of the parent process. If this flag is set, the effective user ID of the child process 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 effective group ID of the parent process. If this flag is set, the effective group ID of the child process 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.

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).

PS 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 systems, but with some POSIX-specific additions.

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 job control of the child process 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 1003.1-200x 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 1003.1-200x 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*( ), *fstatat*( ), *kill*( ), *open*( ), *posix_spawn_file_actions_addclose*( ), *posix_spawn_file_actions_adddup2*( ), *posix_spawn_file_actions_addopen*( ), *posix_spawn_file_actions_destroy*( ), *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_setsigdefault*( ), *posix_spawnattr_setflags*( ), *posix_spawnattr_setpgroup*( ), *posix_spawnattr_setschedparam*( ), *posix_spawnattr_setschedpolicy*( ), *posix_spawnattr_setsigmask*( ), *sched_setparam*( ), *sched_setscheduler*( ), *setpgid*( ), *setuid*( ), *times*( ),

wait( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<spawn.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

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.

Functionality relating to the Threads option is moved to the Base.

NAME

posix_spawn_file_actions_addclose, posix_spawn_file_actions_addopen — add close or open action to spawn file actions object (**ADVANCED REALTIME**)

SYNOPSIS

SPN

```
#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 *
    restrict 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 1003.1-200x 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.

30422 **File Descriptor Mapping**

30423 The standard developers had originally proposed using an array which specified the mapping of
30424 child file descriptors back to those of the parent. It was pointed out by the ballot group that it is
30425 not possible to reshuffle file descriptors arbitrarily in a library implementation of *posix_spawn*( )
30426 or *posix_spawnp*( ) without provision for one or more spare file descriptor entries (which simply
30427 may not be available). Such an array requires that an implementation develop a complex
30428 strategy to achieve the desired mapping without inadvertently closing the wrong file descriptor
30429 at the wrong time.

30430 It was noted by a member of the Ada Language Bindings working group that the approved Ada
30431 Language *Start_Process* family of POSIX process primitives use a caller-specified set of file
30432 actions to alter the normal *fork*( )/*exec* semantics for inheritance of file descriptors in a very
30433 flexible way, yet no such problems exist because the burden of determining how to achieve the
30434 final file descriptor mapping is completely on the application. Furthermore, although the file
30435 actions interface appears frightening at first glance, it is actually quite simple to implement in
30436 either a library or the kernel.

30437 **FUTURE DIRECTIONS**

30438 None.

30439 **SEE ALSO**

30440 *close*( ), *dup*( ), *open*( ), *posix_spawn*( ), *posix_spawn_file_actions_adddup2*( ),
30441 *posix_spawn_file_actions_destroy*( ), *posix_spawnp*( ), the Base Definitions volume of
30442 IEEE Std 1003.1-200x, **<spawn.h>**

30443 **CHANGE HISTORY**

30444 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

30445 IEEE PASC Interpretation 1003.1 #105 is applied, adding a note to the DESCRIPTION that the
30446 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

SPN

```
#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 1003.1-200x, **<spawn.h>**

30487 **CHANGE HISTORY**

30488 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

30489 IEEE PASC Interpretation 1003.1 #104 is applied, noting that the [EBADF] error can apply to the
30490 *newfildes* argument in addition to *fildes*.

30491 **NAME**

30492 posix_spawn_file_actions_addopen — add open action to spawn file actions object
30493 (**ADVANCED REALTIME**)

30494 **SYNOPSIS**

30495 SPN
```
#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);
```

30499 **DESCRIPTION**

30500 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

SPN

```
#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 1003.1-200x, **<spawn.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

30541 **NAME**

30542 posix_spawnattr_destroy, posix_spawnattr_init — destroy and initialize spawn attributes object
30543 (**ADVANCED REALTIME**)

30544 **SYNOPSIS**

30545 SPN
```
#include <spawn.h>

int posix_spawnattr_destroy(posix_spawnattr_t *attr);
int posix_spawnattr_init(posix_spawnattr_t *attr);
```

30548 **DESCRIPTION**

30549 The *posix_spawnattr_destroy*( ) function shall destroy a spawn attributes object. A destroyed *attr*
30550 attributes object can be reinitialized using *posix_spawnattr_init*( ); the results of otherwise
30551 referencing the object after it has been destroyed are undefined. An implementation may cause
30552 *posix_spawnattr_destroy*( ) to set the object referenced by *attr* to an invalid value.

30553 The *posix_spawnattr_init*( ) function shall initialize a spawn attributes object *attr* with the default
30554 value for all of the individual attributes used by the implementation. Results are undefined if
30555 *posix_spawnattr_init*( ) is called specifying an already initialized *attr* attributes object.

30556 A spawn attributes object is of type **posix_spawnattr_t** (defined in **<spawn.h>**) and is used to
30557 specify the inheritance of process attributes across a spawn operation. IEEE Std 1003.1-200x does
30558 not define comparison or assignment operators for the type **posix_spawnattr_t**.

30559 Each implementation shall document the individual attributes it uses and their default values
30560 unless these values are defined by IEEE Std 1003.1-200x. Attributes not defined by
30561 IEEE Std 1003.1-200x, their default values, and the names of the associated functions to get and
30562 set those attribute values are implementation-defined.

30563 The resulting spawn attributes object (possibly modified by setting individual attribute values),
30564 is used to modify the behavior of *posix_spawn*( ) or *posix_spawnp*( ). After a spawn attributes
30565 object has been used to spawn a process by a call to a *posix_spawn*( ) or *posix_spawnp*( ), any
30566 function affecting the attributes object (including destruction) shall not affect any process that
30567 has been spawned in this way.

30568 **RETURN VALUE**

30569 Upon successful completion, *posix_spawnattr_destroy*( ) and *posix_spawnattr_init*( ) shall return
30570 zero; otherwise, an error number shall be returned to indicate the error.

30571 **ERRORS**

30572 The *posix_spawnattr_init*( ) function shall fail if:

30573 [ENOMEM] Insufficient memory exists to initialize the spawn attributes object.

30574 The *posix_spawnattr_destroy*( ) function may fail if:

30575 [EINVAL] The value specified by attr is invalid.

30576 **EXAMPLES**

30577 None.

30578 **APPLICATION USAGE**

30579 These functions are part of the Spawn option and need not be provided on all implementations.

30580 **RATIONALE**

30581 The original spawn interface proposed in IEEE Std 1003.1-200x defined the attributes that
30582 specify the inheritance of process attributes across a spawn operation as a structure. In order to
30583 be able to separate optional individual attributes under their appropriate options (that is, the
30584 *spawn-schedparam* and *spawn-schedpolicy* attributes depending upon the Process Scheduling
30585 option), and also for extensibility and consistency with the newer POSIX interfaces, the
30586 attributes interface has been changed to an opaque data type. This interface now consists of the
30587 type **posix_spawnattr_t**, representing a spawn attributes object, together with associated
30588 functions to initialize or destroy the attributes object, and to set or get each individual attribute.
30589 Although the new object-oriented interface is more verbose than the original structure, it is
30590 simple to use, more extensible, and easy to implement.

30591 **FUTURE DIRECTIONS**

30592 None.

30593 **SEE ALSO**

30594 *posix_spawn( )*, *posix_spawnattr_getsigdefault( )*, *posix_spawnattr_getflags( )*,
30595 *posix_spawnattr_getpgroup( )*, *posix_spawnattr_getschedparam( )*, *posix_spawnattr_getschedpolicy( )*,
30596 *posix_spawnattr_getsigmask( )*, *posix_spawnattr_setsigdefault( )*, *posix_spawnattr_setflags( )*,
30597 *posix_spawnattr_setpgroup( )*, *posix_spawnattr_setsigmask( )*, *posix_spawnattr_setschedpolicy( )*,
30598 *posix_spawnattr_setschedparam( )*, *posix_spawnp( )*, the Base Definitions volume of
30599 IEEE Std 1003.1-200x, **<spawn.h>**

30600 **CHANGE HISTORY**

30601 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

30602 IEEE PASC Interpretation 1003.1 #106 is applied, noting that the effect of initializing an already
30603 initialized spawn attributes option is undefined.

30604 **NAME**

30605 posix_spawnattr_getflags, posix_spawnattr_setflags — get and set the spawn-flags attribute of a
30606 spawn attributes object (**ADVANCED REALTIME**)

30607 **SYNOPSIS**

30608 SPN
```
#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);
```

30612 **DESCRIPTION**

30613 The *posix_spawnattr_getflags*( ) function shall obtain the value of the *spawn-flags* attribute from the
30614 attributes object referenced by *attr*.

30615 The *posix_spawnattr_setflags*( ) function shall set the *spawn-flags* attribute in an initialized
30616 attributes object referenced by *attr*.

30617 The *spawn-flags* attribute is used to indicate which process attributes are to be changed in the
30618 new process image when invoking *posix_spawn*( ) or *posix_spawnp*( ). It is the bitwise-inclusive
30619 OR of zero or more of the following flags:

30620 POSIX_SPAWN_RESETIDS
30621 POSIX_SPAWN_SETPGROUP
30622 POSIX_SPAWN_SETSIGDEF
30623 POSIX_SPAWN_SETSIGMASK
30624 PS POSIX_SPAWN_SETSCHEDPARAM
30625 POSIX_SPAWN_SETSCHEDULER

30626 These flags are defined in **<spawn.h>**. The default value of this attribute shall be as if no flags
30627 were set.

30628 **RETURN VALUE**

30629 Upon successful completion, *posix_spawnattr_getflags*( ) shall return zero and store the value of
30630 the *spawn-flags* attribute of *attr* into the object referenced by the *flags* parameter; otherwise, an
30631 error number shall be returned to indicate the error.

30632 Upon successful completion, *posix_spawnattr_setflags*( ) shall return zero; otherwise, an error
30633 number shall be returned to indicate the error.

30634 **ERRORS**

30635 These functions may fail if:

30636 [EINVAL] The value specified by *attr* is invalid.

30637 The *posix_spawnattr_setflags*( ) function may fail if:

30638 [EINVAL] The value of the attribute being set is not valid.

30639 **EXAMPLES**

30640 None.

30641 **APPLICATION USAGE**

30642 These functions are part of the Spawn option and need not be provided on all implementations.

30643 **RATIONALE**

30644 None.

30645 **FUTURE DIRECTIONS**

30646 None.

30647 **SEE ALSO**

30648 *posix_spawn( )*, *posix_spawnattr_destroy( )*, *posix_spawnattr_init( )*, *posix_spawnattr_getsigdefault( )*,
30649 *posix_spawnattr_getpgroup( )*, *posix_spawnattr_getschedparam( )*, *posix_spawnattr_getschedpolicy( )*,
30650 *posix_spawnattr_getsigmask( )*, *posix_spawnattr_setsigdefault( )*, *posix_spawnattr_setpgroup( )*,
30651 *posix_spawnattr_setschedparam( )*, *posix_spawnattr_setschedpolicy( )*, *posix_spawnattr_setsigmask( )*,
30652 *posix_spawnp( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<spawn.h>**

30653 **CHANGE HISTORY**

30654 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

NAME

posix_spawnattr_getpgroup, posix_spawnattr_setpgroup — get and set the spawn-pgroup attribute of a spawn attributes object (**ADVANCED REALTIME**)

SYNOPSIS

SPN

```
#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_setflags*( ), *posix_spawnattr_setschedparam*( ), *posix_spawnattr_setschedpolicy*( ), *posix_spawnattr_setsigmask*( ), *posix_spawnp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<spawn.h>**

30696 **CHANGE HISTORY**
30697 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

30698 **NAME**

30699 posix_spawnattr_getschedparam, posix_spawnattr_setschedparam — get and set the spawn-
30700 schedparam attribute of a spawn attributes object (**ADVANCED REALTIME**)

30701 **SYNOPSIS**

30702 SPN PS
```
#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);
```

30708 **DESCRIPTION**

30709 The *posix_spawnattr_getschedparam*( ) function shall obtain the value of the *spawn-schedparam*
30710 attribute from the attributes object referenced by *attr*.

30711 The *posix_spawnattr_setschedparam*( ) function shall set the *spawn-schedparam* attribute in an
30712 initialized attributes object referenced by *attr*.

30713 The *spawn-schedparam* attribute represents the scheduling parameters to be assigned to the new
30714 process image in a spawn operation (if POSIX_SPAWN_SETSCHEDULER or
30715 POSIX_SPAWN_SETSCHEDPARAM is set in the *spawn-flags* attribute). The default value of this
30716 attribute is unspecified.

30717 **RETURN VALUE**

30718 Upon successful completion, *posix_spawnattr_getschedparam*( ) shall return zero and store the
30719 value of the *spawn-schedparam* attribute of *attr* into the object referenced by the *schedparam*
30720 parameter; otherwise, an error number shall be returned to indicate the error.

30721 Upon successful completion, *posix_spawnattr_setschedparam*( ) shall return zero; otherwise, an
30722 error number shall be returned to indicate the error.

30723 **ERRORS**

30724 These functions may fail if:

30725 [EINVAL] The value specified by *attr* is invalid.

30726 The *posix_spawnattr_setschedparam*( ) function may fail if:

30727 [EINVAL] The value of the attribute being set is not valid.

30728 **EXAMPLES**

30729 None.

30730 **APPLICATION USAGE**

30731 These functions are part of the Spawn and Process Scheduling options and need not be provided
30732 on all implementations.

30733 **RATIONALE**

30734 None.

30735 **FUTURE DIRECTIONS**

30736 None.

30737 **SEE ALSO**

30738 *posix_spawn( )*, *posix_spawnattr_destroy( )*, *posix_spawnattr_init( )*, *posix_spawnattr_getsigdefault( )*,
30739 *posix_spawnattr_getflags( )*, *posix_spawnattr_getpgroup( )*, *posix_spawnattr_getschedpolicy( )*,
30740 *posix_spawnattr_getsigmask( )*, *posix_spawnattr_setsigdefault( )*, *posix_spawnattr_setflags( )*,
30741 *posix_spawnattr_setpgroup( )*, *posix_spawnattr_setschedpolicy( )*, *posix_spawnattr_setsigmask( )*,
30742 *posix_spawnp( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<sched.h>**, **<spawn.h>**

30743 **CHANGE HISTORY**

30744 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

NAME

posix_spawnattr_getschedpolicy, posix_spawnattr_setschedpolicy — get and set the spawn-schedpolicy attribute of a spawn attributes object (**ADVANCED REALTIME**)

SYNOPSIS

SPN PS

```
#include <spawn.h>
#include <sched.h>

int posix_spawnattr_getschedpolicy(const posix_spawnattr_t *
    restrict 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.

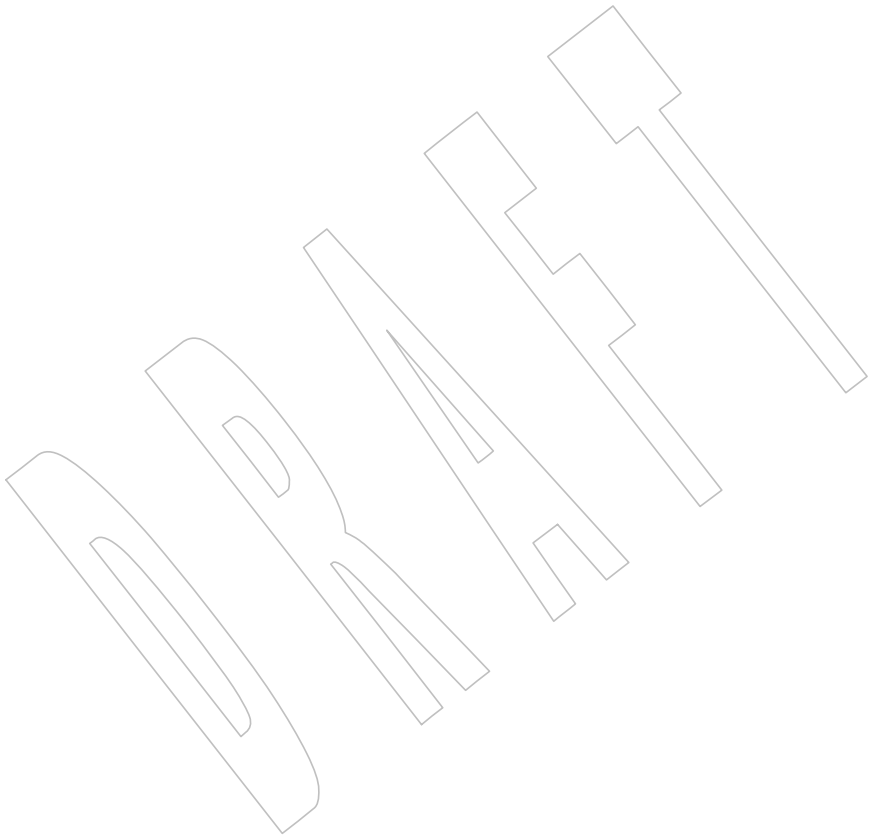
SEE ALSO

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*( ),

30788 *posix_spawnp( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<sched.h>**, **<spawn.h>**

30789 **CHANGE HISTORY**

30790 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.



NAME

posix_spawnattr_getsigdefault, posix_spawnattr_setsigdefault — get and set the spawn-sigdefault attribute of a spawn attributes object (**ADVANCED REALTIME**)

SYNOPSIS

SPN

```
#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 1003.1-200x, **<signal.h>**, **<spawn.h>**

30834 **CHANGE HISTORY**
30835 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

NAME

posix_spawnattr_getsigmask, posix_spawnattr_setsigmask — get and set the spawn-sigmask attribute of a spawn attributes object (**ADVANCED REALTIME**)

SYNOPSIS

SPN

```
#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 1003.1-200x, **<signal.h>**, **<spawn.h>**

30879 **CHANGE HISTORY**

30880 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

30881 **NAME**

30882 posix_spawnattr_init — initialize the spawn attributes object (**ADVANCED REALTIME**)

30883 **SYNOPSIS**

30884 SPN
```
#include <spawn.h>
```
30885
```
int posix_spawnattr_init(posix_spawnattr_t *attr);
```

30886 **DESCRIPTION**

30887 Refer to *posix_spawnattr_destroy( )*.

30888 **NAME**

30889 posix_spawnattr_setflags — set the spawn-flags attribute of a spawn attributes object
30890 (**ADVANCED REALTIME**)

30891 **SYNOPSIS**

30892 SPN
```
#include <spawn.h>

int posix_spawnattr_setflags(posix_spawnattr_t *attr, short flags);
```

30894 **DESCRIPTION**

30895 Refer to *posix_spawnattr_getflags( )*.

30896 **NAME**

30897 posix_spawnattr_setpgroup — set the spawn-pgroup attribute of a spawn attributes object
30898 (**ADVANCED REALTIME**)

30899 **SYNOPSIS**

30900 SPN
```
#include <spawn.h>

int posix_spawnattr_setpgroup(posix_spawnattr_t *attr, pid_t pgroup);
```

30902 **DESCRIPTION**

30903 Refer to *posix_spawnattr_getpgroup( )*.

30904 **NAME**

30905 posix_spawnattr_setschedparam — set the spawn-schedparam attribute of a spawn attributes
30906 object (**ADVANCED REALTIME**)

30907 **SYNOPSIS**

30908 SPN PS
```
#include <sched.h>
#include <spawn.h>

int posix_spawnattr_setschedparam(posix_spawnattr_t *restrict attr,
    const struct sched_param *restrict schedparam);
```

30912 **DESCRIPTION**

30913 Refer to *posix_spawnattr_getschedparam( )*.

30914 **NAME**

30915 posix_spawnattr_setschedpolicy — set the spawn-schedpolicy attribute of a spawn attributes
30916 object (**ADVANCED REALTIME**)

30917 **SYNOPSIS**

30918 SPN PS
```
#include <sched.h>
#include <spawn.h>

int posix_spawnattr_setschedpolicy(posix_spawnattr_t *attr,
    int schedpolicy);
```

30922 **DESCRIPTION**

30923 Refer to *posix_spawnattr_getschedpolicy( )*.

30924 **NAME**

30925 posix_spawnattr_setsigdefault — set the spawn-sigdefault attribute of a spawn attributes object
30926 (**ADVANCED REALTIME**)

30927 **SYNOPSIS**

30928 SPN
```
#include <signal.h>
#include <spawn.h>

int posix_spawnattr_setsigdefault(posix_spawnattr_t *restrict attr,
    const sigset_t *restrict sigdefault);
```

30932 **DESCRIPTION**

30933 Refer to *posix_spawnattr_getsigdefault( )*.

30934 **NAME**

30935 posix_spawnattr_setsigmask — set the spawn-sigmask attribute of a spawn attributes object
30936 (**ADVANCED REALTIME**)

30937 **SYNOPSIS**

30938 SPN
```
#include <signal.h>
#include <spawn.h>

int posix_spawnattr_setsigmask(posix_spawnattr_t *restrict attr,
    const sigset_t *restrict sigmask);
```

30942 **DESCRIPTION**

30943 Refer to *posix_spawnattr_getsigmask( )*.

30944 **NAME**

30945 posix_spawnp — spawn a process (**ADVANCED REALTIME**)

30946 **SYNOPSIS**

30947 SPN

```
#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]);
```

30952 **DESCRIPTION**

30953 Refer to *posix_spawn( )*.

NAME

posix_trace_attr_destroy, posix_trace_attr_init — destroy and initialize the trace stream attributes object (**TRACING**)

SYNOPSIS

OB TRC

```
#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 77)).

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 1003.1-200x, 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.

30994 **FUTURE DIRECTIONS**

30995 The *posix_trace_attr_destroy*( ) and *posix_trace_attr_init*( ) functions may be removed in a future
30996 version.

30997 **SEE ALSO**

30998 *posix_trace_create*( ), *posix_trace_get_attr*( ), *uname*( ), the Base Definitions volume of
30999 IEEE Std 1003.1-200x, **<trace.h>**

31000 **CHANGE HISTORY**

31001 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

31002 IEEE PASC Interpretation 1003.1 #123 is applied.

31003 **Issue 7**

31004 The *posix_trace_attr_destroy*( ) and *posix_trace_attr_init*( ) functions are marked obsolescent.

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

OB 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

The *posix_trace_attr_getclockres*( ), *posix_trace_attr_getcreatetime*( ), *posix_trace_attr_getgenversion*( ), *posix_trace_attr_getname*( ), and *posix_trace_attr_setname*( ) functions may be removed in a future version.

SEE ALSO

posix_trace_attr_init( ), *posix_trace_create*( ), *posix_trace_get_attr*( ), *uname*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**, **<trace.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

Issue 7

The *posix_trace_attr_getclockres*( ), *posix_trace_attr_getcreatetime*( ), *posix_trace_attr_getgenversion*( ), *posix_trace_attr_getname*( ), and *posix_trace_attr_setname*( ) functions are marked obsolescent.

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

OB TRC
```
#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 *restrict
    attr, int *restrict 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

TRI 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 **<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.

TRL 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.

TRL 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.

TRL 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.

TRI 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.

TRL 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.

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

The *posix_trace_attr_getinherited*( ), *posix_trace_attr_getlogfullpolicy*( ), *posix_trace_attr_getstreamfullpolicy*( ), *posix_trace_attr_setinherited*( ), *posix_trace_attr_setlogfullpolicy*( ), and *posix_trace_attr_setstreamfullpolicy*( ) functions may be removed in a future version.

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 1003.1-200x, **<trace.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/39 is applied, adding the TRL and TRC margin codes to the *posix_trace_attr_setlogfullpolicy*( ) function.

Issue 7

SD5-XSH-ERN-116 is applied, adding the missing **restrict** keyword to the *posix_trace_attr_getstreamfullpolicy*( ) function declaration.

These functions are marked obsolescent.

NAME

posix_trace_attr_getlogsize, posix_trace_attr_getmaxdatasize, posix_trace_attr_getmaxsystemeventsize, posix_trace_attr_getmaxuserevent size, 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

OB TRC
```
#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_getmaxuserevent size(
    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

TRL 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 *stream-min-size* attribute of that trace stream, the system provides the necessary resources for recording all those trace events, without loss.

The *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 *posix_trace_event*( ) with a *data_len* parameter equal to the *data_len* value specified in this call. 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 *posix_trace_attr_getstreamsize*( ) function shall copy the stream size, in bytes, from the *stream-min-size* attribute of the attributes object pointed to by the *attr* argument into the variable pointed to by the *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 *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 *posix_trace_attr_setmaxdatasize*( ) function shall set the maximum allowed size, in bytes, in the *max-data-size* attribute of the attributes object pointed to by the *attr* argument, using the size value supplied by the *maxdatasize* argument. This maximum size is the maximum allowed size for the user data argument which may be passed to *posix_trace_event*( ). The implementation shall be allowed to truncate data passed to *trace_user_event* which is longer than *maxdatasize*.

The *posix_trace_attr_setstreamsize*( ) function shall set the minimum allowed size, in bytes, in the *stream-min-size* attribute of the attributes object pointed to by the *attr* argument, using the size value supplied by the *streamsize* argument.

RETURN VALUE

Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

TRL The *posix_trace_attr_getlogsize*( ) function stores the maximum trace log allowed size in the object pointed to by *logsize*, if successful.

The *posix_trace_attr_getmaxdatasize*( ) function stores the maximum trace event record memory size in the object pointed to by *maxdatasize*, if successful.

The *posix_trace_attr_getmaxsystemeventsize*( ) function stores the maximum memory size to store a single system trace event in the object pointed to by *eventsize*, if successful.

The *posix_trace_attr_getmaxusereventsize*( ) function stores the maximum memory size to store a single user trace event in the object pointed to by *eventsize*, if successful.

The *posix_trace_attr_getstreamsize*( ) function stores the maximum trace stream allowed size in the object pointed to by *streamsize*, if successful.

ERRORS

These functions may fail if:

[EINVAL] The value specified by one of the arguments is invalid.

31283 **EXAMPLES**

31284 None.

31285 **APPLICATION USAGE**

31286 None.

31287 **RATIONALE**

31288 None.

31289 **FUTURE DIRECTIONS**

31290 The *posix_trace_attr_getlogsize*( ), *posix_trace_attr_getmaxdatasize*( ),
31291 *posix_trace_attr_getmaxsystemeventsize*( ), *posix_trace_attr_getmaxuserevent*size( ),
31292 *posix_trace_attr_getstreamsize*( ), *posix_trace_attr_setlogsize*( ), *posix_trace_attr_setmaxdatasize*( ), and
31293 *posix_trace_attr_setstreamsize*( ) functions may be withdrawn in a future version.

31294 **SEE ALSO**

31295 *posix_trace_attr_init*( ), *posix_trace_create*( ), *posix_trace_event*( ), *posix_trace_get_attr*( ), the Base
31296 Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<trace.h>**

31297 **CHANGE HISTORY**

31298 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

31299 **Issue 7**

31300 These functions are marked obsolescent.

posix_trace_attr_getname( )

31301 **NAME**

31302 posix_trace_attr_getname — retrieve and set information about a trace stream (**TRACING**)

31303 **SYNOPSIS**

31304 OB TRC
```
#include <trace.h>

int posix_trace_attr_getname(const trace_attr_t *attr,
    char *tracename);
```

31307 **DESCRIPTION**

31308 Refer to *posix_trace_attr_getclockres( )*.

31309 **NAME**

31310 posix_trace_attr_getstreamfullpolicy — retrieve and set the behavior of a trace stream
31311 (**TRACING**)

31312 **SYNOPSIS**

31313 OB TRC
```
#include <trace.h>

int posix_trace_attr_getstreamfullpolicy(const trace_attr_t *restrict
    attr, int *restrict streampolicy);
```

31316 **DESCRIPTION**

31317 Refer to *posix_trace_attr_getinherited( )*.

31318 **NAME**

31319 posix_trace_attr_getstreamsize — retrieve and set trace stream size attributes (**TRACING**)

31320 **SYNOPSIS**

31321 OB TRC
```
#include <sys/types.h>
#include <trace.h>

int posix_trace_attr_getstreamsize(const trace_attr_t *restrict attr,
    size_t *restrict streamsize);
```

31325 **DESCRIPTION**

31326 Refer to *posix_trace_attr_getlogsize( )*.

31327 **NAME**

31328 posix_trace_attr_init — initialize the trace stream attributes object (**TRACING**)

31329 **SYNOPSIS**

31330 OB TRC
```
#include <trace.h>

int posix_trace_attr_init(trace_attr_t *attr);
```

31332 **DESCRIPTION**

31333 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

OB TRC
```
#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( )*.

31345 **NAME**

31346 posix_trace_attr_setlogsize, posix_trace_attr_setmaxdatasize — retrieve and set trace stream size
31347 attributes (**TRACING**)

31348 **SYNOPSIS**

31349 OB TRC
```
#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);
```

31355 **DESCRIPTION**

31356 Refer to *posix_trace_attr_getlogsize( )*.

posix_trace_attr_setname( )

31357 **NAME**

31358 posix_trace_attr_setname — retrieve and set information about a trace stream (**TRACING**)

31359 **SYNOPSIS**

31360 OB TRC
```
#include <trace.h>

int posix_trace_attr_setname(trace_attr_t *attr,
    const char *tracename);
```

31363 **DESCRIPTION**

31364 Refer to *posix_trace_attr_getclockres( )*.

31365 **NAME**

31366 posix_trace_attr_setstreamfullpolicy — retrieve and set the behavior of a trace stream
31367 (**TRACING**)

31368 **SYNOPSIS**

31369 OB TRC
```
#include <trace.h>

int posix_trace_attr_setstreamfullpolicy(trace_attr_t *attr,
    int streampolicy);
```

31372 **DESCRIPTION**

31373 Refer to *posix_trace_attr_getinherited( )*.

posix_trace_attr_setstreamsize( )

NAME

posix_trace_attr_setstreamsize — retrieve and set trace stream size attributes (**TRACING**)

SYNOPSIS

OB TRC

```
#include <sys/types.h>
#include <trace.h>

int posix_trace_attr_setstreamsize(trace_attr_t *attr,
    size_t streamsize);
```

DESCRIPTION

Refer to

NAME

posix_trace_clear — clear trace stream and trace log (**TRACING**)

SYNOPSIS

OB TRC

```
#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.

OB TRL

If the Trace Log option is supported and the trace stream has been created with a log, the *posix_trace_clear*( ) function shall reinitialize the trace stream with the same behavior as if the trace stream was created without the log, plus it shall reinitialize the trace log associated with the trace stream identified by the argument *trid* as if it were returning from the *posix_trace_create_withlog*( ) function, except that the same allocated resources, for the trace log, may be reused and the associated trace stream status remains unchanged. The first trace event recorded in the trace log after the call to *posix_trace_clear*( ) shall be the same as the first trace event recorded in the active trace stream after the call to *posix_trace_clear*( ). The *posix_log_full_status* status shall be set to POSIX_TRACE_NOT_FULL. There is no guarantee that all trace events that occurred during the *posix_trace_clear*( ) call are recorded in the trace log; the behavior with respect to trace points that may occur during this call is unspecified. If the log full policy is POSIX_TRACE_APPEND, the effect of a call to this function is unspecified for the trace log associated with the trace stream identified by the *trid* argument.

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

The *posix_trace_clear*( ) function may be withdrawn in a future version.

SEE ALSO

posix_trace_attr_init( ), *posix_trace_create*( ), *posix_trace_flush*( ), *posix_trace_get_attr*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<trace.h>**

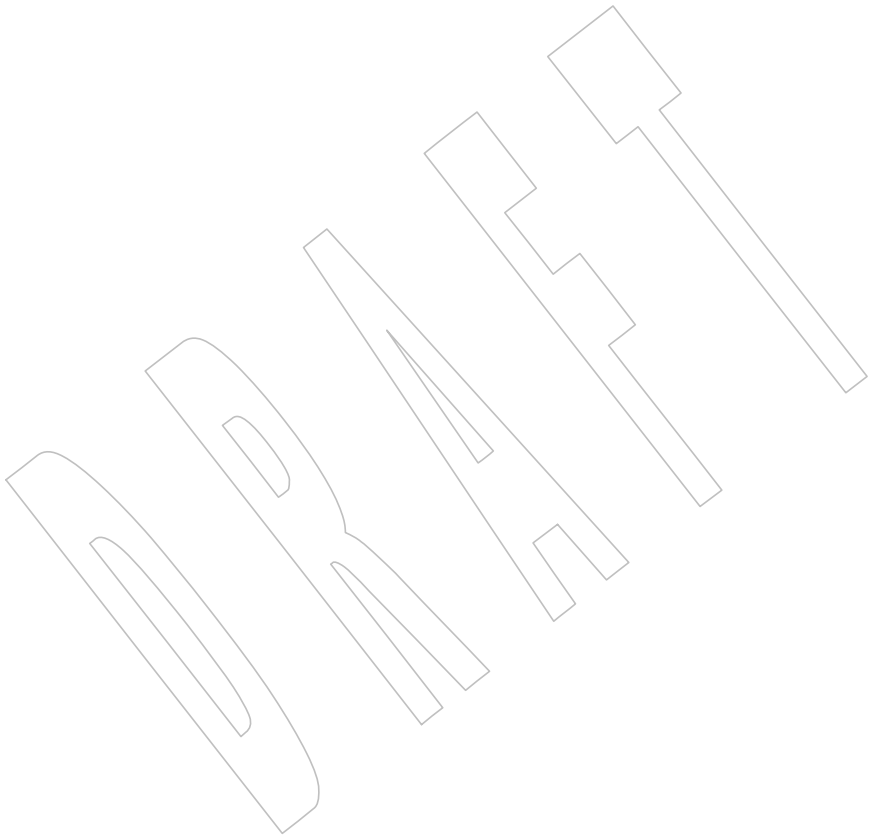
CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

IEEE PASC Interpretation 1003.1 #123 is applied.

Issue 7

The *posix_trace_clear*( ) function is marked obsolescent.



NAME

posix_trace_close, posix_trace_open, posix_trace_rewind — trace log management (**TRACING**)

SYNOPSIS

OB TRC
```
#include <trace.h>
```

TRL
```
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_eventid_equal( )
posix_trace_eventid_get_name( )
posix_trace_eventtypelist_getnext_id( )
posix_trace_eventtypelist_rewind( )
posix_trace_get_attr( )
posix_trace_get_status( )
posix_trace_getnext_event( )
posix_trace_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

The *posix_trace_close*( ), *posix_trace_open*( ), and *posix_trace_rewind*( ) functions may be removed in a future version.

SEE ALSO

posix_trace_get_attr( ), *posix_trace_get_filter*( ), *posix_trace_getnext_event*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<trace.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

IEEE PASC Interpretation 1003.1 #123 is applied.

Issue 7

The *posix_trace_close*( ), *posix_trace_open*( ), and *posix_trace_rewind*( ) functions are marked obsolescent.

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

OB TRC
```
#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);
```
TRL
```
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*( )

TEF 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.

TEF 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 1003.1-200x, these functions shall return with the error [EINVAL].

TRL 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_clear( )
posix_trace_eventid_equal( )
posix_trace_eventid_get_name( )
posix_trace_eventtypelist_getnext_id( )
posix_trace_eventtypelist_rewind( )
posix_trace_flush( )
posix_trace_get_attr( )
posix_trace_get_status( )
posix_trace_getnext_event( )
posix_trace_shutdown( )
posix_trace_start( )
posix_trace_stop( )
posix_trace_timedgetnext_event( )
posix_trace_trid_eventid_open( )

TEF TRL 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*( )

TRL 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_withlog*( ) function.

The *posix_trace_flush*( ) function shall initiate a flush operation which copies the contents of the trace stream identified by the argument *trid* into the trace log associated with the trace stream at the creation time. If no trace log has been associated with the trace stream pointed to by *trid*, this function shall return an error. The termination of the flush operation can be polled by the *posix_trace_get_status*( ) function. During the flush operation, it shall be possible to trace new trace events up to the point when the trace stream becomes full. After flushing is completed, the

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.

TRL 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.

TRL 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

TRL 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.

[ENOMEM] The implementation does not currently have sufficient memory to create the trace stream with the specified parameters.

[EPERM] The caller does not have appropriate privilege to trace the process specified by *pid*.

[ESRCH] The *pid* argument does not refer to an existing process.

TRL The *posix_trace_create_withlog*( ) function shall fail if:

[EBADF] The *file_desc* argument is not a valid file descriptor open for writing.

[EINVAL]	The *file_desc* argument refers to a file with a file type that does not support the log policy associated with the trace log.
[ENOSPC]	No space left on device. The device corresponding to the argument *file_desc* does not contain the space required to create this trace log.

TRL The *posix_trace_flush*( ) and *posix_trace_shutdown*( ) functions shall fail if:

[EINVAL]	The value of the *trid* argument does not correspond to an active trace stream with log.
[EFBIG]	The trace log file has attempted to exceed an implementation-defined maximum file size.
[ENOSPC]	No space left on device.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

The *posix_trace_create*( ), *posix_trace_create_withlog*( ), *posix_trace_flush*( ), and *posix_trace_shutdown*( ) functions may be withdrawn in a future version.

SEE ALSO

clock_getres( ), *exec*, *posix_trace_attr_init*( ), *posix_trace_clear*( ), *posix_trace_close*( ), *posix_trace_eventid_equal*( ), *posix_trace_eventtypelist_getnext_id*( ), *posix_trace_flush*( ), *posix_trace_get_attr*( ), *posix_trace_get_filter*( ), *posix_trace_get_status*( ), *posix_trace_getnext_event*( ), *posix_trace_open*( ), *posix_trace_set_filter*( ), *posix_trace_shutdown*( ), *posix_trace_start*( ), *posix_trace_timedgetnext_event*( ), *posix_trace_trid_eventid_open*( ), *posix_trace_start*( ), *time*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<trace.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

Issue 7

These functions are marked obsolescent.

SD5-XSH-ERN-154 is applied, updating the DESCRIPTION to remove the *posix_trace_trygetnext_event*( ) function from the list of functions that use the *trid* argument.

NAME

posix_trace_event, posix_trace_eventid_open — trace functions for instrumenting application code (**TRACING**)

SYNOPSIS

OB TRC

```
#include <sys/types.h>
#include <trace.h>

void posix_trace_event(trace_event_id_t event_id,
    const void *restrictdata_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_USEREVENT (see Table 2-7 (on page 81)) user trace event shall be returned.

TRI 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 81)) 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

The *posix_trace_event*( ) and *posix_trace_eventid_open*( ) functions may be withdrawn in a future version.

SEE ALSO

Table 2-7 (on page 81), *exec*, *posix_trace_start*( ), *posix_trace_trid_eventid_open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<trace.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

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.

Issue 7

The *posix_trace_event*( ) and *posix_trace_eventid_open*( ) functions are marked obsolescent.

NAME

posix_trace_eventid_equal, posix_trace_eventid_get_name, posix_trace_trid_eventid_open — manipulate the trace event type identifier (**TRACING**)

SYNOPSIS

OB TRC
```
#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);
```
TEF
```
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.

TEF 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 81)) user trace event shall be returned.

TEF TRI 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_UNNAMED_USEREVENT (see Table 2-7 (on page 81)) user trace event shall be returned.

RETURN VALUE

TEF 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.

TEF The *posix_trace_trid_eventid_open*( ) function stores the trace event type identifier value in the object pointed to by *event*, if successful.

ERRORS

TEF 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.

TEF The *posix_trace_trid_eventid_open*( ) function shall fail if:

TEF [ENAMETOOLONG]
The size of the name pointed to by the *event_name* argument was longer than the implementation-defined value {TRACE_EVENT_NAME_MAX}.

The *posix_trace_eventid_get_name*( ) function shall fail if:

[EINVAL] The trace event type identifier *event* was not associated with any name.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

The *posix_trace_eventid_equal*( ), *posix_trace_eventid_get_name*( ), and *posix_trace_trid_eventid_open*( ) functions may be withdrawn in a future version.

SEE ALSO

Table 2-7 (on page 81), *exec*, *posix_trace_event*( ), *posix_trace_getnext_event*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<trace.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

IEEE PASC Interpretations 1003.1 #123 and #129 are applied.

Issue 7

These functions are marked obsolescent.

31825 **NAME**

31826 posix_trace_eventid_open — trace functions for instrumenting application code (**TRACING**)

31827 **SYNOPSIS**

31828 OB TRC
```
#include <sys/types.h>
#include <trace.h>

int posix_trace_eventid_open(const char *restrict event_name,
    trace_event_id_t *restrict event_id);
```

31832 **DESCRIPTION**

31833 Refer to *posix_trace_event( )*.

NAME

posix_trace_eventset_add, posix_trace_eventset_del, posix_trace_eventset_empty, posix_trace_eventset_fill, posix_trace_eventset_ismember — manipulate trace event type sets (**TRACING**)

SYNOPSIS

OB TRC
TEF

```
#include <trace.h>

int posix_trace_eventset_add(trace_event_id_t event_id,
    trace_event_set_t *set);
int posix_trace_eventset_del(trace_event_id_t event_id,
    trace_event_set_t *set);
int posix_trace_eventset_empty(trace_event_set_t *set);
int posix_trace_eventset_fill(trace_event_set_t *set, int what);
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 1003.1-200x.

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.

31879 **RETURN VALUE**

31880 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
31881 return the corresponding error number.

31882 **ERRORS**

31883 These functions may fail if:

31884 [EINVAL] The value of one of the arguments is invalid.

31885 **EXAMPLES**

31886 None.

31887 **APPLICATION USAGE**

31888 None.

31889 **RATIONALE**

31890 None.

31891 **FUTURE DIRECTIONS**

31892 The *posix_trace_eventset_add*( ), *posix_trace_eventset_del*( ), *posix_trace_eventset_empty*( ),
31893 *posix_trace_eventset_fill*( ), and *posix_trace_eventset_ismember*( ) functions may be removed in a
31894 future version.

31895 **SEE ALSO**

31896 *posix_trace_set_filter*( ), *posix_trace_trid_eventid_open*( ), the Base Definitions volume of
31897 IEEE Std 1003.1-200x, **<trace.h>**

31898 **CHANGE HISTORY**

31899 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

31900 **Issue 7**

31901 The *posix_trace_eventset_add*( ), *posix_trace_eventset_del*( ), *posix_trace_eventset_empty*( ),
31902 *posix_trace_eventset_fill*( ), and *posix_trace_eventset_ismember*( ) functions are marked obsolescent.

NAME

posix_trace_eventtypelist_getnext_id, posix_trace_eventtypelist_rewind — iterate over a mapping of trace event types (**TRACING**)

SYNOPSIS

OB TRC

```
#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:

[EINVAL] The *trid* argument was not a valid trace stream identifier.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

The *posix_trace_eventtypelist_getnext_id*( ) and *posix_trace_eventtypelist_rewind*( ) functions may be removed in a future version.

SEE ALSO

posix_trace_event( ), *posix_trace_getnext_event*( ), *posix_trace_trid_eventid_open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<trace.h>**

31944 **CHANGE HISTORY**

31945 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

31946 IEEE PASC Interpretations 1003.1 #123 and #129 are applied.

31947 **Issue 7**

31948 The *posix_trace_eventtypelist_getnext_id*( ) and *posix_trace_eventtypelist_rewind*( ) functions are
31949 marked obsolescent.

31950 **NAME**

31951 posix_trace_flush — trace stream flush from a process (**TRACING**)

31952 **SYNOPSIS**

31953 OB TRC
```
#include <sys/types.h>
```
31954
```
#include <trace.h>
```
31955 TRL
```
int posix_trace_flush(trace_id_t trid);
```

31956 **DESCRIPTION**

31957 Refer to *posix_trace_create( )*.

NAME

posix_trace_get_attr, posix_trace_get_status — retrieve the trace attributes or trace status (**TRACING**)

SYNOPSIS

OB TRC
```
#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. TRL 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. TRL 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. TRL 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.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

The *posix_trace_get_attr*( ) and *posix_trace_get_status*( ) functions may be withdrawn in a future version.

SEE ALSO

posix_trace_attr_destroy( ), *posix_trace_attr_init*( ), *posix_trace_create*( ), *posix_trace_open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<trace.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

IEEE PASC Interpretation 1003.1 #123 is applied.

Issue 7

The *posix_trace_get_attr*( ) and *posix_trace_get_status*( ) functions are marked obsolescent.

NAME

posix_trace_get_filter, posix_trace_set_filter — retrieve and set the filter of an initialized trace stream (**TRACING**)

SYNOPSIS

OB TRC
```
#include <trace.h>
```

TEF
```
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 and Table 2-6 (on page 80)).

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.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

The *posix_trace_get_filter*( ) and *posix_trace_set_filter*( ) functions may be removed in a future version.

SEE ALSO

Table 2-4 (on page 79), Table 2-6 (on page 80), *posix_trace_eventset_add*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<trace.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

IEEE PASC Interpretation 1003.1 #123 is applied.

Issue 7

The *posix_trace_get_filter*( ) and *posix_trace_set_filter*( ) functions are marked obsolescent.

32074 **NAME**

32075 posix_trace_get_status — retrieve the trace status (**TRACING**)

32076 **SYNOPSIS**

32077 OB TRC
```
#include <trace.h>

int posix_trace_get_status(trace_id_t trid,
    struct posix_trace_status_info *statusinfo);
```

32080 **DESCRIPTION**

32081 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

OB TRC
```
#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 TRL 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.

The timeout shall be based on the CLOCK_REALTIME clock. 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

a trace event is immediately available from the trace stream.

The behavior of this function for a pre-recorded trace stream is unspecified.

TRL 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_timedgetnext_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 *timeout* expired.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

The *posix_trace_getnext_event*( ), *posix_trace_timedgetnext_event*( ), and *posix_trace_trygetnext_event*( ) functions may be removed in a future version.

SEE ALSO

posix_trace_create( ), *posix_trace_open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<trace.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

IEEE PASC Interpretation 1003.1 #123 is applied.

Issue 7

The *posix_trace_getnext_event*( ), *posix_trace_timedgetnext_event*( ), and *posix_trace_trygetnext_event*( ) functions are marked obsolescent.

Functionality relating to the Timers option is moved to the Base.

32195 **NAME**

32196 posix_trace_open, posix_trace_rewind — trace log management (**TRACING**)

32197 **SYNOPSIS**

32198 OB TRC
```
#include <trace.h>
```

32199 TRL
```
int posix_trace_open(int file_desc, trace_id_t *trid);
```
32200
```
int posix_trace_rewind(trace_id_t trid);
```

32201 **DESCRIPTION**

32202 Refer to *posix_trace_close( )*.

NAME

posix_trace_set_filter — set filter of an initialized trace stream (**TRACING**)

SYNOPSIS

OB TRC TEF

```
#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( )*.

32211 **NAME**

32212 posix_trace_shutdown — trace stream shutdown from a process (**TRACING**)

32213 **SYNOPSIS**

32214 OB TRC
```
#include <sys/types.h>
#include <trace.h>

int posix_trace_shutdown(trace_id_t trid);
```

32217 **DESCRIPTION**

32218 Refer to *posix_trace_create( )*.

NAME

posix_trace_start, posix_trace_stop — trace start and stop (**TRACING**)

SYNOPSIS

OB TRC

```
#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.

[EINTR] The operation was interrupted by a signal and thus the trace stream was not necessarily started or stopped.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

The *posix_trace_start*( ) and *posix_trace_stop*( ) functions may be removed in a future version.

SEE ALSO

posix_trace_create( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<trace.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

IEEE PASC Interpretation 1003.1 #123 is applied.

Issue 7

The *posix_trace_start*( ) and *posix_trace_stop*( ) functions are marked obsolescent.

posix_trace_timedgetnext_event( )

32264 **NAME**

32265 posix_trace_timedgetnext_event — retrieve a trace event (**TRACING**)

32266 **SYNOPSIS**

32267 OB TRC
```
#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);
```

32274 **DESCRIPTION**

32275 Refer to *posix_trace_getnext_event( )*.

32276 **NAME**

32277 posix_trace_trid_eventid_open — open a trace event type identifier (**TRACING**)

32278 **SYNOPSIS**

32279 OB TRC
```
#include <trace.h>
```

32280 TEF
```
int posix_trace_trid_eventid_open(trace_id_t trid,
    const char *restrict event_name,
    trace_event_id_t *restrict event);
```

32283 **DESCRIPTION**

32284 Refer to *posix_trace_eventid_equal( )*.

posix_trace_trygetnext_event( )

NAME

posix_trace_trygetnext_event — retrieve a trace event (**TRACING**)

SYNOPSIS

OB TRC

```
#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

TYM
```
#include <sys/mman.h>

int posix_typed_mem_get_info(int fildes,
    struct posix_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

allocated to prevent race conditions.

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 1003.1-200x, **<sys/mman.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

NAME

posix_typed_mem_open — open a typed memory object (**ADVANCED REALTIME**)

SYNOPSIS

TYM
```
#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 1003.1-200x 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 allocation; that is, mapping such an object leaves each byte of the mapped area unallocated if it was unallocated prior to the mapping or allocated if it was allocated prior to the mapping. The appropriate privilege to specify the POSIX_TYPED_MEM_MAP_ALLOCATABLE flag is implementation-defined.

If successful, *posix_typed_mem_open*( ) shall return a file descriptor for the typed memory object that is the lowest numbered file descriptor not currently open for that process. The open file description is new, and therefore the file descriptor shall not share it with any other processes. It is unspecified whether the file offset is set. The FD_CLOEXEC file descriptor flag associated with the new file descriptor shall be cleared.

The behavior of *msync*( ), *ftruncate*( ), and all file operations other than *mmap*( ), *posix_mem_offset*( ), *posix_typed_mem_get_info*( ), *fstat*( ), *dup*( ), *dup2*( ), and *close*( ), is unspecified when passed a file descriptor connected to a typed memory object by this function.

The file status flags of the open file description shall be set according to the value of *oflag*. Applications shall specify exactly one of the three access mode values described below and defined in the **<fcntl.h>** header, as the value of *oflag*.

O_RDONLY Open for read access only.

O_WRONLY Open for write access only.

O_RDWR Open for read or write access.

RETURN VALUE

Upon successful completion, the *posix_typed_mem_open*( ) function shall return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

The *posix_typed_mem_open*( ) function shall fail if:

[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_MEM_ALLOCATE_CONTIG, or POSIX_TYPED_MEM_MAP_ALLOCATABLE is specified).

[EMFILE] All file descriptors available to the process are currently open.

[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*.

32439 **EXAMPLES**

32440 None.

32441 **APPLICATION USAGE**

32442 None.

32443 **RATIONALE**

32444 None.

32445 **FUTURE DIRECTIONS**

32446 None.

32447 **SEE ALSO**

32448 *close( )*, *dup( )*, *exec*, *fcntl( )*, *fstat( )*, *ftruncate( )*, *mmap( )*, *msync( )*, *posix_mem_offset( )*,
32449 *posix_typed_mem_get_info( )*, *umask( )*, the Base Definitions volume of IEEE Std 1003.1-200x,
32450 **<fcntl.h>**, **<sys/mman.h>**

32451 **CHANGE HISTORY**

32452 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

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

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 1003.1-200x 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 *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 value of x raised to the power y.

MX 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 *pow*( ), *powf*( ), and *powl*( ) shall return ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL, respectively, with the same sign as the correct value of the function.

If the correct value would cause underflow, and is not representable, a range error may occur, and MX either 0.0 (if supported), or an implementation-defined value shall be returned.

CX For $y < 0$, if x is zero, a pole error may occur and *pow*( ), *powf*( ), and *powl*( ) shall return MX ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL, respectively. On systems that support the IEC 60559 Floating-Point option, a pole error shall occur and *pow*( ), *powf*( ), and *powl*( ) shall return ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL, respectively if y is an odd integer, or HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively if y is not an odd integer.

MX 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.

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.

MX 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:

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 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.

32530 **EXAMPLES**

32531 None.

32532 **APPLICATION USAGE**

32533 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
32534 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

32535 **RATIONALE**

32536 None.

32537 **FUTURE DIRECTIONS**

32538 None.

32539 **SEE ALSO**

32540 *exp( )*, *feclearexcept( )*, *fetestexcept( )*, *isnan( )*, the Base Definitions volume of IEEE Std 1003.1-200x,
32541 Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

32542 **CHANGE HISTORY**

32543 First released in Issue 1. Derived from Issue 1 of the SVID.

32544 **Issue 5**

32545 The DESCRIPTION is updated to indicate how an application should check for an error. This
32546 text was previously published in the APPLICATION USAGE section.

32547 **Issue 6**

32548 The normative text is updated to avoid use of the term "must" for application requirements.

32549 The *powf( )* and *powl( )* functions are added for alignment with the ISO/IEC 9899: 1999 standard.

32550 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
32551 revised to align with the ISO/IEC 9899: 1999 standard.

32552 IEC 60559: 1989 standard floating-point extensions over the ISO/IEC 9899: 1999 standard are
32553 marked.

32554 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/42 is applied, correcting the third
32555 paragraph in the RETURN VALUE section.

32556 **Issue 7**

32557 ISO/IEC 9899: 1999 standard, Technical Corrigendum 2 #51 (SD5-XSH-ERN-81) is applied.

32558 **NAME**

32559 pread — read from a file

32560 **SYNOPSIS**

32561
```
#include <unistd.h>
```

32562
```
ssize_t pread(int fildes, void *buf, size_t nbyte, off_t offset);
```

32563 **DESCRIPTION**

32564 Refer to *read( )*.

32565 **NAME**

32566 printf — print formatted output

32567 **SYNOPSIS**

32568
```
#include <stdio.h>
```

32569
```
int printf(const char *restrict format, ...);
```

32570 **DESCRIPTION**

32571 Refer to *fprintf( )*.

32572 **NAME**

32573 pselect, select — synchronous I/O multiplexing

32574 **SYNOPSIS**

32575
```
#include <sys/select.h>
```

32576
```
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);
```

32587 **DESCRIPTION**

32588 The *pselect*( ) function shall examine the file descriptor sets whose addresses are passed in the
32589 *readfds*, *writefds*, and *errorfds* parameters to see whether some of their descriptors are ready for
32590 reading, are ready for writing, or have an exceptional condition pending, respectively.

32591 The *select*( ) function shall be equivalent to the *pselect*( ) function, except as follows:

32592 • For the *select*( ) function, the timeout period is given in seconds and microseconds in an
32593 argument of type **struct timeval**, whereas for the *pselect*( ) function the timeout period is
32594 given in seconds and nanoseconds in an argument of type **struct timespec**.

32595 • The *select*( ) function has no *sigmask* argument; it shall behave as *pselect*( ) does when
32596 *sigmask* is a null pointer.

32597 • Upon successful completion, the *select*( ) function may modify the object pointed to by the
32598 *timeout* argument.

32599 The *pselect*( ) and *select*( ) functions shall support regular files, terminal and pseudo-terminal
32600 OB XSR devices, STREAMS-based files, FIFOs, pipes, and sockets. The behavior of *pselect*( ) and *select*( )
32601 on file descriptors that refer to other types of file is unspecified.

32602 The *nfds* argument specifies the range of descriptors to be tested. The first *nfds* descriptors shall
32603 be checked in each set; that is, the descriptors from zero through *nfds*–1 in the descriptor sets
32604 shall be examined.

32605 If the *readfds* argument is not a null pointer, it points to an object of type **fd_set** that on input
32606 specifies the file descriptors to be checked for being ready to read, and on output indicates
32607 which file descriptors are ready to read.

32608 If the *writefds* argument is not a null pointer, it points to an object of type **fd_set** that on input
32609 specifies the file descriptors to be checked for being ready to write, and on output indicates
32610 which file descriptors are ready to write.

32611 If the *errorfds* argument is not a null pointer, it points to an object of type **fd_set** that on input
32612 specifies the file descriptors to be checked for error conditions pending, and on output indicates
32613 which file descriptors have error conditions pending.

32614 Upon successful completion, the *pselect*( ) or *select*( ) function shall modify the objects pointed to
32615 by the *readfds*, *writefds*, and *errorfds* arguments to indicate which file descriptors are ready for
32616 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*( ) 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 caller by the set of signals pointed to by *sigmask* before examining the descriptors, and shall restore the signal mask of the calling thread 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 marked as writable.

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.

If a thread gets canceled during a *pselect*( ) call, the signal mask in effect when executing the registered cleanup functions is either the original signal mask or the signal mask installed as part of the *pselect*( ) call.

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.

32710 **ERRORS**

32711 Under the following conditions, *pselect*( ) and *select*( ) shall fail and set *errno* to:

32712 [EBADF] One or more of the file descriptor sets specified a file descriptor that is not a
32713 valid open file descriptor.

32714 [EINTR] The function was interrupted before any of the selected events occurred and
32715 before the timeout interval expired.

32716 XSI If SA_RESTART has been set for the interrupting signal, it is implementation-
32717 defined whether the function restarts or returns with [EINTR].

32718 [EINVAL] An invalid timeout interval was specified.

32719 [EINVAL] The *nfds* argument is less than 0 or greater than FD_SETSIZE.

32720 OB XSR [EINVAL] One of the specified file descriptors refers to a STREAM or multiplexer that is
32721 linked (directly or indirectly) downstream from a multiplexer.

32722 **EXAMPLES**

32723 None.

32724 **APPLICATION USAGE**

32725 None.

32726 **RATIONALE**

32727 In previous versions of the Single UNIX Specification, the *select*( ) function was defined in the
32728 **<sys/time.h>** header. This is now changed to **<sys/select.h>**. The rationale for this change was
32729 as follows: the introduction of the *pselect*( ) function included the **<sys/select.h>** header and the
32730 **<sys/select.h>** header defines all the related definitions for the *pselect*( ) and *select*( ) functions.
32731 Backwards-compatibility to existing XSI implementations is handled by allowing **<sys/time.h>**
32732 to include **<sys/select.h>**.

32733 Code which wants to avoid the ambiguity of the signal mask for thread cancellation handlers
32734 can install an additional cancellation handler which resets the signal mask to the expected value.

```
32735 void cleanup(void *arg)
32736 {
32737     sigset_t *ss = (sigset_t *) arg;
32738     pthread_sigmask(SIG_SETMASK, ss, NULL);
32739 }

32740 int call_pselect(int nfds, fd_set *readfds, fd_set *writefds,
32741     fd_set errorfds, const struct timespec *timeout,
32742     const sigset_t *sigmask)
32743 {
32744     sigset_t oldmask;
32745     int result;
32746     pthread_sigmask(SIG_SETMASK, NULL, &oldmask);
32747     pthread_cleanup_push(cleanup, &oldmask);
32748     result = pselect(nfds, readfds, writefds, errorfds, timeout, sigmask)
32749     pthread_cleanup_pop(0);
32750     return result;
32751 }
```

32752 **FUTURE DIRECTIONS**

32753 None.

32754 **SEE ALSO**

32755 *accept( )*, *alarm( )*, *connect( )*, *fcntl( )*, *poll( )*, *read( )*, *recvmsg( )*, *sendmsg( )*, *setitimer( )*, *write( )*, the
32756 Base Definitions volume of IEEE Std 1003.1-200x, **<sys/select.h>**, **<sys/time.h>**

32757 **CHANGE HISTORY**

32758 First released in Issue 4, Version 2.

32759 **Issue 5**

32760 Moved from X/OPEN UNIX extension to BASE.

32761 In the ERRORS section, the text has been changed to indicate that [EINVAL] is returned when
32762 *nfds* is less than 0 or greater than FD_SETSIZE. It previously stated less than 0, or greater than or
32763 equal to FD_SETSIZE.

32764 Text about *timeout* is moved from the APPLICATION USAGE section to the DESCRIPTION.

32765 **Issue 6**

32766 The Open Group Corrigendum U026/6 is applied, changing the occurrences of *readfs* and *writefs*
32767 in the *select( )* DESCRIPTION to be *readfds* and *writefds*.

32768 Text referring to sockets is added to the DESCRIPTION.

32769 The DESCRIPTION and ERRORS sections are updated so that references to STREAMS are
32770 marked as part of the XSI STREAMS Option Group.

32771 The following new requirements on POSIX implementations derive from alignment with the
32772 Single UNIX Specification:

32773 - These functions are now mandatory.

32774 The *pselect( )* function is added for alignment with IEEE Std 1003.1g-2000 and additional detail
32775 related to sockets semantics is added to the DESCRIPTION.

32776 The *select( )* function now requires inclusion of **<sys/select.h>**.

32777 The **restrict** keyword is added to the *select( )* prototype for alignment with the
32778 ISO/IEC 9899: 1999 standard.

32779 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/70 is applied, updating the
32780 DESCRIPTION to reference the signal mask in terms of the calling thread rather than the
32781 process.

32782 **Issue 7**

32783 SD5-XSH-ERN-122 is applied, adding text to the DESCRIPTION for when a thread is canceled
32784 during a call to *pselect( )*, and adding example code to the RATIONALE.

32785 Functionality relating to the XSI STREAMS option is marked obsolescent.

32786 Functionality relating to the Threads option is moved to the Base.

NAME

psiginfo, psignal — print signal information to standard error

SYNOPSIS

CX

```
#include <signal.h>

void psiginfo(siginfo_t *pinfo, const char *message);
void psignal(int signum, const char *message);
```

DESCRIPTION

The *psiginfo*( ) and *psignal*( ) functions shall print a message out on *stderr* associated with a signal number. If *message* is not null and is not the empty string, then the string pointed to by the *message* argument shall be printed first, followed by a colon, a space, and the signal description string indicated by *signum*, or by the signal associated with *pinfo*. If the *message* argument is null or points to an empty string, then only the signal description shall be printed. For *psiginfo*( ), the argument *pinfo* references a valid **siginfo_t** structure. For *psignal*( ), if *signum* is not a valid signal number, the behavior is implementation-defined.

RETURN VALUE

These functions shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

System V historically has *psignal*( ) and *psiginfo*( ) in **<siginfo.h>**. However, the **<siginfo.h>** header is not specified in the Base Definitions volume of IEEE Std 1003.1-200x, and the type **siginfo_t** is defined in **<signal.h>**.

FUTURE DIRECTIONS

None.

SEE ALSO

perror( ), *strsignal*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

CHANGE HISTORY

First released in Issue 7.

32819 **NAME**

32820 psignal — print signal information to standard error

32821 **SYNOPSIS**

32822 CX
```
#include <signal.h>

void psignal(int signum, const char *message);
```

32824 **DESCRIPTION**

32825 Refer to *psiginfo( )*.

NAME

pthread_atfork — register fork handlers

SYNOPSIS

```
#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( ), *exec*, *fork*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

Issue 7

The *pthread_atfork*( ) function is moved from the Threads option to the Base.

NAME

pthread_attr_destroy, pthread_attr_init — destroy and initialize the thread attributes object

SYNOPSIS

```
#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.

The *pthread_attr_destroy*( ) function may fail if:

[EINVAL] The value specified by *attr* does not refer to an initialized thread attribute object.

The *pthread_attr_init*( ) function may fail if:

[EBUSY] The implementation has detected an attempt to reinitialize the thread attribute referenced by *attr*, a previously initialized, but not yet destroyed, thread attribute.

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 1003.1-200x 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_getstacksize( ), *pthread_attr_getdetachstate*( ), *pthread_create*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/71 is applied, updating the ERRORS section to add the optional [EINVAL] error for the *pthread_attr_destroy*( ) function, and the optional [EBUSY] error for the *pthread_attr_init*( ) function.

Issue 7

The *pthread_attr_destroy*( ) and *pthread_attr_init*( ) functions are moved from the Threads option to the Base.

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 may fail if:

[EINVAL] The value specified by *attr* does not refer to an initialized thread attribute object.

These functions shall not return an error code of [EINTR].

EXAMPLES

Retrieving the detachstate Attribute

This example shows how to obtain the *detachstate* attribute of a thread attribute object.

```
#include <pthread.h>

pthread_attr_t thread_attr;
int            detachstate;
int            rc;

/* code initializing thread_attr */
```



```
...
rc = pthread_attr_getdetachstate (&thread_attr, &detachstate);
if (rc!=0) {
    /* handle error */
    ...
}
else {
    /* legal values for detachstate are:
     * PTHREAD_CREATE_DETACHED or PTHREAD_CREATE_JOINABLE
     */
     ...
}
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_attr_destroy( ), *pthread_attr_getstacksize( )*, *pthread_create( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<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 normative text is updated to avoid use of the term "must" for application requirements.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/72 is applied, adding the example to the EXAMPLES section.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/73 is applied, updating the ERRORS section to include the optional [EINVAL] error.

Issue 7

The *pthread_attr_setdetachstate*( ) and *pthread_attr_getdetachstate*( ) functions are moved from the Threads option to the Base.

NAME

pthread_attr_getguardsize, pthread_attr_setguardsize — get and set the thread guardsize attribute

SYNOPSIS

```
#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* 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

These functions shall fail if:

[EINVAL] The parameter *guardsize* is invalid.

These functions may fail if:

[EINVAL] The value specified by *attr* does not refer to an initialized thread attribute object.

These functions shall not return an error code of [EINTR].

EXAMPLES

Retrieving the guardsize Attribute

This example shows how to obtain the *guardsize* attribute of a thread attribute object.

```
#include <pthread.h>

pthread_attr_t thread_attr;
size_t  guardsize;
int     rc;

/* code initializing thread_attr */
...

rc = pthread_attr_getguardsize (&thread_attr, &guardsize);
if (rc != 0)  {
    /* handle error */
    ...
}
else {
    if (guardsize > 0) {
    /* a guard area of at least guardsize bytes is provided */
    ...
    }
    else {
    /* no guard area provided */
    ...
    }
}
```

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 1003.1-200x, **<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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/74 is applied, updating the ERRORS section to remove the [EINVAL] error (“The attribute *attr* is invalid.”), and replacing it with the

optional [EINVAL] error.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/76 is applied, adding the example to the EXAMPLES section.

Issue 7

SD5-XSH-ERN-111 is applied, removing the reference to the *stack* attribute in the DESCRIPTION.

The *pthread_attr_getguardsize*( ) and *pthread_attr_setguardsize*( ) functions are moved from the XSI option to the Base.

NAME

pthread_attr_getinheritsched, pthread_attr_setinheritsched — get and set the inheritsched attribute (**REALTIME THREADS**)

SYNOPSIS

TPS

```
#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.

The supported values of *inheritsched* shall be:

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 1003.1-200x 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.

ERRORS

The *pthread_attr_getinheritsched*( ) function may fail if:

[EINVAL] The value specified by *attr* does not refer to an initialized thread attribute object.

The *pthread_attr_setinheritsched*( ) function may fail if:

[EINVAL] The value of *inheritsched* is not valid.

[EINVAL] The value specified by *attr* does not refer to an initialized thread attribute object.

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

These functions shall not return an error code of [EINTR].

33236 **EXAMPLES**

33237 None.

33238 **APPLICATION USAGE**

33239 After these attributes have been set, a thread can be created with the specified attributes using
33240 *pthread_create*( ). Using these routines does not affect the current running thread.

33241 See Section 2.9.4 for further details on thread scheduling attributes and their default settings.

33242 **RATIONALE**

33243 None.

33244 **FUTURE DIRECTIONS**

33245 None.

33246 **SEE ALSO**

33247 *pthread_attr_destroy*( ), *pthread_attr_getscope*( ), *pthread_attr_getschedpolicy*( ),
33248 *pthread_attr_getschedparam*( ), *pthread_create*( ), the Base Definitions volume of
33249 IEEE Std 1003.1-200x, **<pthread.h>**, **<sched.h>**

33250 **CHANGE HISTORY**

33251 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33252 Marked as part of the Realtime Threads Feature Group.

33253 **Issue 6**

33254 The *pthread_attr_getinheritsched*( ) and *pthread_attr_setinheritsched*( ) functions are marked as part
33255 of the Threads and Thread Execution Scheduling options.

33256 The [ENOSYS] error condition has been removed as stubs need not be provided if an
33257 implementation does not support the Thread Execution Scheduling option.

33258 The **restrict** keyword is added to the *pthread_attr_getinheritsched*( ) prototype for alignment with
33259 the ISO/IEC 9899: 1999 standard.

33260 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/75 is applied, clarifying the values of
33261 *inheritsched* in the DESCRIPTION and adding two optional [EINVAL] errors to the ERRORS
33262 section for checking when *attr* refers to an uninitialized thread attribute object.

33263 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/77 is applied, adding a reference to
33264 Section 2.9.4 in the APPLICATION USAGE section.

33265 **Issue 7**

33266 The *pthread_attr_getinheritsched*( ) and *pthread_attr_setinheritsched*( ) functions are moved from the
33267 Threads option.

33268 **NAME**

33269 pthread_attr_getschedparam, pthread_attr_setschedparam — get and set the schedparam
33270 attribute

33271 **SYNOPSIS**

33272
```
#include <pthread.h>
```

33273
```
int pthread_attr_getschedparam(const pthread_attr_t *restrict attr,
    struct sched_param *restrict param);
```
33274
33275
```
int pthread_attr_setschedparam(pthread_attr_t *restrict attr,
    const struct sched_param *restrict param);
```
33276

33277 **DESCRIPTION**

33278 The *pthread_attr_getschedparam*( ), and *pthread_attr_setschedparam*( ) functions, respectively, shall
33279 get and set the scheduling parameter attributes in the *attr* argument. The contents of the *param*
33280 structure are defined in the **<sched.h>** header. For the SCHED_FIFO and SCHED_RR policies,
33281 the only required member of *param* is *sched_priority*.

33282 TSP For the SCHED_SPORADIC policy, the required members of the *param* structure are
33283 *sched_priority*, *sched_ss_low_priority*, *sched_ss_repl_period*, *sched_ss_init_budget*, and
33284 *sched_ss_max_repl*. The specified *sched_ss_repl_period* must be greater than or equal to the
33285 specified *sched_ss_init_budget* for the function to succeed; if it is not, then the function shall fail.
33286 The value of *sched_ss_max_repl* shall be within the inclusive range [1,{SS_REPL_MAX}] for the
33287 function to succeed; if not, the function shall fail.

33288 **RETURN VALUE**

33289 If successful, the *pthread_attr_getschedparam*( ) and *pthread_attr_setschedparam*( ) functions shall
33290 return zero; otherwise, an error number shall be returned to indicate the error.

33291 **ERRORS**

33292 The *pthread_attr_getschedparam*( ) function may fail if:

33293 [EINVAL] The value specified by *attr* does not refer to an initialized thread attribute
33294 object.

33295 The *pthread_attr_setschedparam*( ) function may fail if:

33296 [EINVAL] The value of *param* is not valid, or the value specified by *attr* does not refer to
33297 an initialized thread attribute object.

33298 [ENOTSUP] An attempt was made to set the attribute to an unsupported value.

33299 These functions shall not return an error code of [EINTR].

33300 **EXAMPLES**

33301 None.

33302 **APPLICATION USAGE**

33303 After these attributes have been set, a thread can be created with the specified attributes using
33304 *pthread_create*( ). Using these routines does not affect the current running thread.

33305 **RATIONALE**

33306 None.

33307 **FUTURE DIRECTIONS**

33308 None.

SEE ALSO

pthread_attr_destroy( ), *pthread_attr_getscope( )*, *pthread_attr_getinheritsched( )*, *pthread_attr_getschedpolicy( )*, *pthread_create( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**, **<sched.h>**

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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/78 is applied, updating the ERRORS section to include optional errors for the case when *attr* refers to an uninitialized thread attribute object.

Issue 7

The *pthread_attr_getschedparam*( ) and *pthread_attr_setschedparam*( ) functions are moved from the Threads option to the Base.

NAME

pthread_attr_getschedpolicy, pthread_attr_setschedpolicy — get and set the schedpolicy attribute (**REALTIME THREADS**)

SYNOPSIS

TPS
```
#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
TSP
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_getschedpolicy*( ) function may fail if:

[EINVAL] The value specified by *attr* does not refer to an initialized thread attribute object.

The *pthread_attr_setschedpolicy*( ) function may fail if:

[EINVAL] The value of *policy* is not valid, or the value specified by *attr* does not refer to an initialized thread attribute object.

[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.

See Section 2.9.4 for further details on thread scheduling attributes and their default settings.

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 1003.1-200x, **<pthread.h>**, **<sched.h>**

33368 **CHANGE HISTORY**

33369 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33370 Marked as part of the Realtime Threads Feature Group.

33371 **Issue 6**

33372 The *pthread_attr_getschedpolicy*( ) and *pthread_attr_setschedpolicy*( ) functions are marked as part of
33373 the Threads and Thread Execution Scheduling options.

33374 The [ENOSYS] error condition has been removed as stubs need not be provided if an
33375 implementation does not support the Thread Execution Scheduling option.

33376 The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

33377 The **restrict** keyword is added to the *pthread_attr_getschedpolicy*( ) prototype for alignment with
33378 the ISO/IEC 9899: 1999 standard.

33379 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/79 is applied, adding a reference to
33380 Section 2.9.4 in the APPLICATION USAGE section.

33381 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/80 is applied, updating the ERRORS
33382 section to include optional errors for the case when *attr* refers to an uninitialized thread attribute
33383 object.

33384 **Issue 7**

33385 The *pthread_attr_getschedpolicy*( ) and *pthread_attr_setschedpolicy*( ) functions are moved from the
33386 Threads option.

NAME

pthread_attr_getscope, pthread_attr_setscope — get and set the contentionscope attribute (**REALTIME THREADS**)

SYNOPSIS

TPS

```
#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_getscope*( ) function may fail if:

[EINVAL] The value specified by *attr* does not refer to an initialized thread attribute object.

The *pthread_attr_setscope*( ) function may fail if:

[EINVAL] The value of *contentionscope* is not valid, or the value specified by *attr* does not refer to an initialized thread attribute object.

[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.

See Section 2.9.4 for further details on thread scheduling attributes and their default settings.

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 1003.1-200x, **<pthread.h>**, **<sched.h>**

33428 **CHANGE HISTORY**

33429 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33430 Marked as part of the Realtime Threads Feature Group.

33431 **Issue 6**

33432 The *pthread_attr_getscope*( ) and *pthread_attr_setscope*( ) functions are marked as part of the
33433 Threads and Thread Execution Scheduling options.

33434 The [ENOSYS] error condition has been removed as stubs need not be provided if an
33435 implementation does not support the Thread Execution Scheduling option.

33436 The **restrict** keyword is added to the *pthread_attr_getscope*( ) prototype for alignment with the
33437 ISO/IEC 9899: 1999 standard.

33438 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/81 is applied, adding a reference to
33439 Section 2.9.4 in the APPLICATION USAGE section.

33440 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/82 is applied, updating the ERRORS
33441 section to include optional errors for the case when *attr* refers to an uninitialized thread attribute
33442 object.

33443 **Issue 7**

33444 The *pthread_attr_getscope*( ) and *pthread_attr_setscope*( ) functions are moved from the Threads
33445 option.

NAME

pthread_attr_getstack, pthread_attr_setstack — get and set stack attributes

SYNOPSIS

TSA TSS

```
#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.

If the *pthread_attr_getstack*( ) function is called before the *stackaddr* attribute has been set, the behavior is unspecified.

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_getstack*( ) function may fail if:

[EINVAL] The value specified by *attr* does not refer to an initialized thread attribute object.

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 (*stackaddr* + *stacksize*) lacks proper alignment, or the value specified by *attr* does not refer to an initialized thread attribute object.

[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].

33484 **EXAMPLES**

33485 None.

33486 **APPLICATION USAGE**

33487 These functions are appropriate for use by applications in an environment where the stack for a
33488 thread must be placed in some particular region of memory.

33489 While it might seem that an application could detect stack overflow by providing a protected
33490 page outside the specified stack region, this cannot be done portably. Implementations are free
33491 to place the thread's initial stack pointer anywhere within the specified region to accommodate
33492 the machine's stack pointer behavior and allocation requirements. Furthermore, on some
33493 architectures, such as the IA-64, "overflow" might mean that two separate stack pointers
33494 allocated within the region will overlap somewhere in the middle of the region.

33495 After a successful call to *pthread_attr_setstack*( ), the storage area specified by the *stackaddr*
33496 parameter is under the control of the implementation, as described in Section 2.9.8 (on page 59).

33497 The specification of the *stackaddr* attribute presents several ambiguities that make portable use of
33498 these functions impossible.

33499 **RATIONALE**

33500 None.

33501 **FUTURE DIRECTIONS**

33502 None.

33503 **SEE ALSO**

33504 *pthread_attr_init*( ), *pthread_attr_setdetachstate*( ), *pthread_attr_setstacksize*( ), *pthread_create*( ), the
33505 Base Definitions volume of IEEE Std 1003.1-200x, **<limits.h>**, **<pthread.h>**

33506 **CHANGE HISTORY**

33507 First released in Issue 6. Developed as part of the XSI option and brought into the BASE by IEEE
33508 PASC Interpretation 1003.1 #101.

33509 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/83 is applied, updating the
33510 APPLICATION USAGE section to refer to Section 2.9.8 (on page 59).

33511 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC/D6/84 is applied, updating the ERRORS
33512 section to include optional errors for the case when *attr* refers to an uninitialized thread attribute
33513 object.

33514 **Issue 7**

33515 SD5-XSH-ERN-66 is applied, correcting the use of *attr* in the [EINVAL] error condition.

33516 Austin Group Interpretation 1003.1-2001 #057 is applied, clarifying the behavior if the function is
33517 called before the *stackaddr* attribute is set.

33518 SD5-XSH-ERN-157 is applied, updating the APPLICATION USAGE section.

NAME

pthread_attr_getstacksize, pthread_attr_setstacksize — get and set the stacksize attribute

SYNOPSIS

TSS

```
#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 may fail if:

[EINVAL] The value specified by *attr* does not refer to an initialized thread attribute object.

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_getdetachstate*( ), *pthread_create*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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 Size Attribute options.

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

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/43 is applied, correcting the margin code in the SYNOPSIS from TSA to TSS and updating the CHANGE HISTORY from "Thread Stack Address Attribute" option to "Thread Stack Size Attribute" option.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/87 is applied, updating the ERRORS section to include optional errors for the case when *attr* refers to an uninitialized thread attribute object.

Issue 7

The *pthread_attr_getstacksize*( ) and *pthread_attr_setstacksize*( ) functions are moved from the Threads option.

pthread_attr_init( )

33571 **NAME**

33572 pthread_attr_init — initialize the thread attributes object

33573 **SYNOPSIS**

33574
```
#include <pthread.h>
```

33575
```
int pthread_attr_init(pthread_attr_t *attr);
```

33576 **DESCRIPTION**

33577 Refer to *pthread_attr_destroy( )*.

33578 **NAME**

33579 pthread_attr_setdetachstate — set the detachstate attribute

33580 **SYNOPSIS**

33581 `#include <pthread.h>`

33582 `int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);`

33583 **DESCRIPTION**

33584 Refer to *pthread_attr_getdetachstate( )*.

NAME

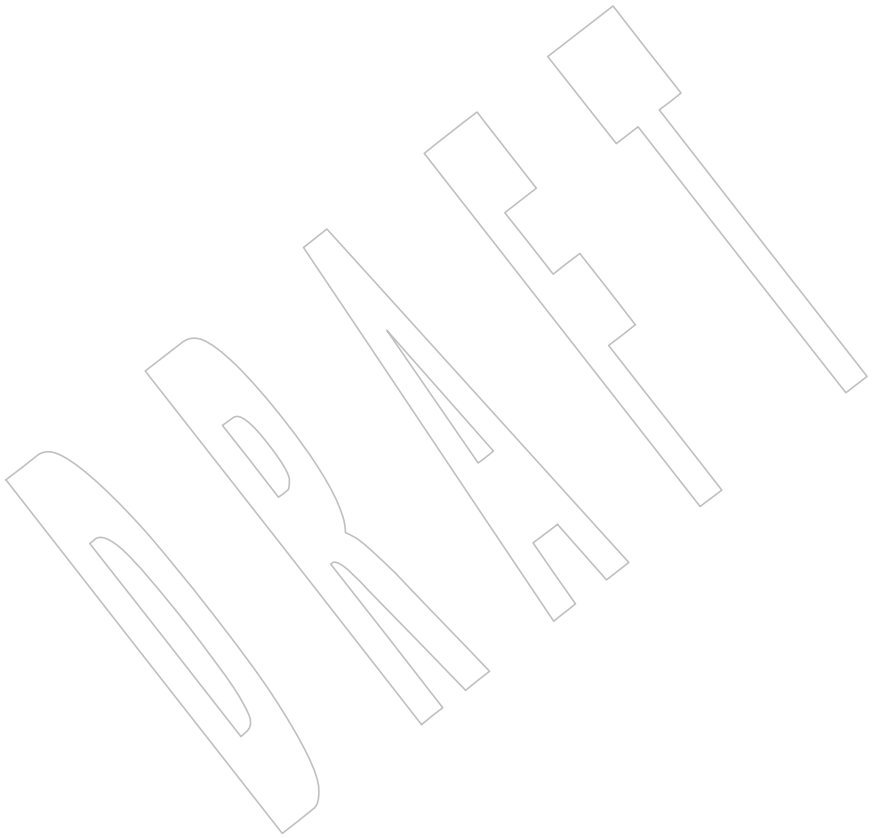
pthread_attr_setguardsize — set the thread guardsize attribute

SYNOPSIS

```
#include <pthread.h>

int pthread_attr_setguardsize(pthread_attr_t *attr,
    size_t guardsize);
```

DESCRIPTION

Refer to *pthread_attr_getguardsize( )*.



33593 **NAME**

33594 pthread_attr_setinheritsched — set the inheritsched attribute (**REALTIME THREADS**)

33595 **SYNOPSIS**

33596 TPS
```
#include <pthread.h>

int pthread_attr_setinheritsched(pthread_attr_t *attr,
    int inheritsched);
```

33599 **DESCRIPTION**

33600 Refer to *pthread_attr_getinheritsched( )*.

33601 **NAME**

33602 pthread_attr_setschedparam — set the schedparam attribute

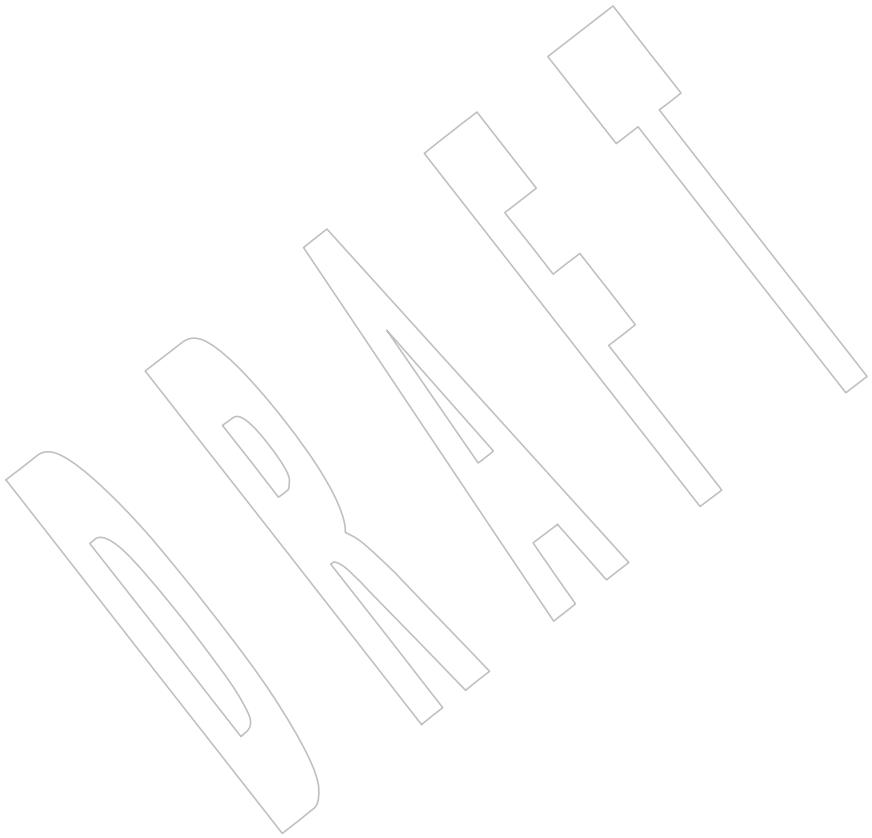
33603 **SYNOPSIS**

33604
```
#include <pthread.h>
```

33605
```
int pthread_attr_setschedparam(pthread_attr_t *restrict attr,
```
33606
```
    const struct sched_param *restrict param);
```

33607 **DESCRIPTION**

33608 Refer to *pthread_attr_getschedparam( )*.



33609 **NAME**

33610 pthread_attr_setschedpolicy — set the schedpolicy attribute (**REALTIME THREADS**)

33611 **SYNOPSIS**

33612 TPS
```
#include <pthread.h>

int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
```

33614 **DESCRIPTION**

33615 Refer to *pthread_attr_getschedpolicy( )*.

pthread_attr_setscope( )

33616 **NAME**

33617 pthread_attr_setscope — set the contentionscope attribute (**REALTIME THREADS**)

33618 **SYNOPSIS**

33619 TPS
```
#include <pthread.h>

int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
```

33620

33621 **DESCRIPTION**

33622 Refer to *pthread_attr_getscope( )*.

33623 **NAME**

33624 pthread_attr_setstack — set the stack attribute

33625 **SYNOPSIS**

33626 TSA TSS
```
#include <pthread.h>

int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr,
    size_t stacksize);
```

33629 **DESCRIPTION**

33630 Refer to *pthread_attr_getstack( )*.

NAME

pthread_attr_setstacksize — set the stacksize attribute

SYNOPSIS

TSS

```
#include <pthread.h>

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

DESCRIPTION

Refer to *pthread_attr_getstacksize( )*.

33638 **NAME**

33639 pthread_barrier_destroy, pthread_barrier_init — destroy and initialize a barrier object

33640 **SYNOPSIS**

33641
```
#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);
```

33645 **DESCRIPTION**

33646 The *pthread_barrier_destroy*( ) function shall destroy the barrier referenced by *barrier* and release
33647 any resources used by the barrier. The effect of subsequent use of the barrier is undefined until
33648 the barrier is reinitialized by another call to *pthread_barrier_init*( ). An implementation may use
33649 this function to set *barrier* to an invalid value. The results are undefined if
33650 *pthread_barrier_destroy*( ) is called when any thread is blocked on the barrier, or if this function is
33651 called with an uninitialized barrier.

33652 The *pthread_barrier_init*( ) function shall allocate any resources required to use the barrier
33653 referenced by *barrier* and shall initialize the barrier with attributes referenced by *attr*. If *attr* is
33654 NULL, the default barrier attributes shall be used; the effect is the same as passing the address of
33655 a default barrier attributes object. The results are undefined if *pthread_barrier_init*( ) is called
33656 when any thread is blocked on the barrier (that is, has not returned from the
33657 *pthread_barrier_wait*( ) call). The results are undefined if a barrier is used without first being
33658 initialized. The results are undefined if *pthread_barrier_init*( ) is called specifying an already
33659 initialized barrier.

33660 The *count* argument specifies the number of threads that must call *pthread_barrier_wait*( ) before
33661 any of them successfully return from the call. The value specified by *count* must be greater than
33662 zero.

33663 If the *pthread_barrier_init*( ) function fails, the barrier shall not be initialized and the contents of
33664 *barrier* are undefined.

33665 Only the object referenced by *barrier* may be used for performing synchronization. The result of
33666 referring to copies of that object in calls to *pthread_barrier_destroy*( ) or *pthread_barrier_wait*( ) is
33667 undefined.

33668 **RETURN VALUE**

33669 Upon successful completion, these functions shall return zero; otherwise, an error number shall
33670 be returned to indicate the error.

33671 **ERRORS**

33672 The *pthread_barrier_destroy*( ) function may fail if:

33673 [EBUSY] The implementation has detected an attempt to destroy a barrier while it is in
33674 use (for example, while being used in a *pthread_barrier_wait*( ) call) by another
33675 thread.

33676 [EINVAL] The value specified by *barrier* is invalid.

33677 The *pthread_barrier_init*( ) function shall fail if:

33678 [EAGAIN] The system lacks the necessary resources to initialize another barrier.

33679 [EINVAL] The value specified by *count* is equal to zero.

[ENOMEM] Insufficient memory exists to initialize the barrier.

The *pthread_barrier_init*( ) function may fail if:

[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

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_barrier_wait( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

Issue 7

The *pthread_barrier_destroy*( ) and *pthread_barrier_init*( ) functions are moved from the Barriers option to the Base.

33702 **NAME**

33703 pthread_barrier_wait — synchronize at a barrier

33704 **SYNOPSIS**

33705 ```
#include <pthread.h>
```

33706 ```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

33707 **DESCRIPTION**

33708 The *pthread_barrier_wait*( ) function shall synchronize participating threads at the barrier
33709 referenced by *barrier*. The calling thread shall block until the required number of threads have
33710 called *pthread_barrier_wait*( ) specifying the barrier.

33711 When the required number of threads have called *pthread_barrier_wait*( ) specifying the barrier,
33712 the constant PTHREAD_BARRIER_SERIAL_THREAD shall be returned to one unspecified
33713 thread and zero shall be returned to each of the remaining threads. At this point, the barrier shall
33714 be reset to the state it had as a result of the most recent *pthread_barrier_init*( ) function that
33715 referenced it.

33716 The constant PTHREAD_BARRIER_SERIAL_THREAD is defined in **<pthread.h>** and its value
33717 shall be distinct from any other value returned by *pthread_barrier_wait*( ).

33718 The results are undefined if this function is called with an uninitialized barrier.

33719 If a signal is delivered to a thread blocked on a barrier, upon return from the signal handler the
33720 thread shall resume waiting at the barrier if the barrier wait has not completed (that is, if the
33721 required number of threads have not arrived at the barrier during the execution of the signal
33722 handler); otherwise, the thread shall continue as normal from the completed barrier wait. Until
33723 the thread in the signal handler returns from it, it is unspecified whether other threads may
33724 proceed past the barrier once they have all reached it.

33725 A thread that has blocked on a barrier shall not prevent any unblocked thread that is eligible to
33726 use the same processing resources from eventually making forward progress in its execution.
33727 Eligibility for processing resources shall be determined by the scheduling policy.

33728 **RETURN VALUE**

33729 Upon successful completion, the *pthread_barrier_wait*( ) function shall return
33730 PTHREAD_BARRIER_SERIAL_THREAD for a single (arbitrary) thread synchronized at the
33731 barrier and zero for each of the other threads. Otherwise, an error number shall be returned to
33732 indicate the error.

33733 **ERRORS**

33734 The *pthread_barrier_wait*( ) function may fail if:

33735 [EINVAL] The value specified by *barrier* does not refer to an initialized barrier object.

33736 This function shall not return an error code of [EINTR].

33737 **EXAMPLES**

33738 None.

33739 **APPLICATION USAGE**

33740 Applications using this function may be subject to priority inversion, as discussed in the Base
33741 Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

33742 **RATIONALE**

33743 None.

33744 **FUTURE DIRECTIONS**

33745 None.

33746 **SEE ALSO**

33747 *pthread_barrier_destroy*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.10,
33748 Memory Synchronization, **<pthread.h>**

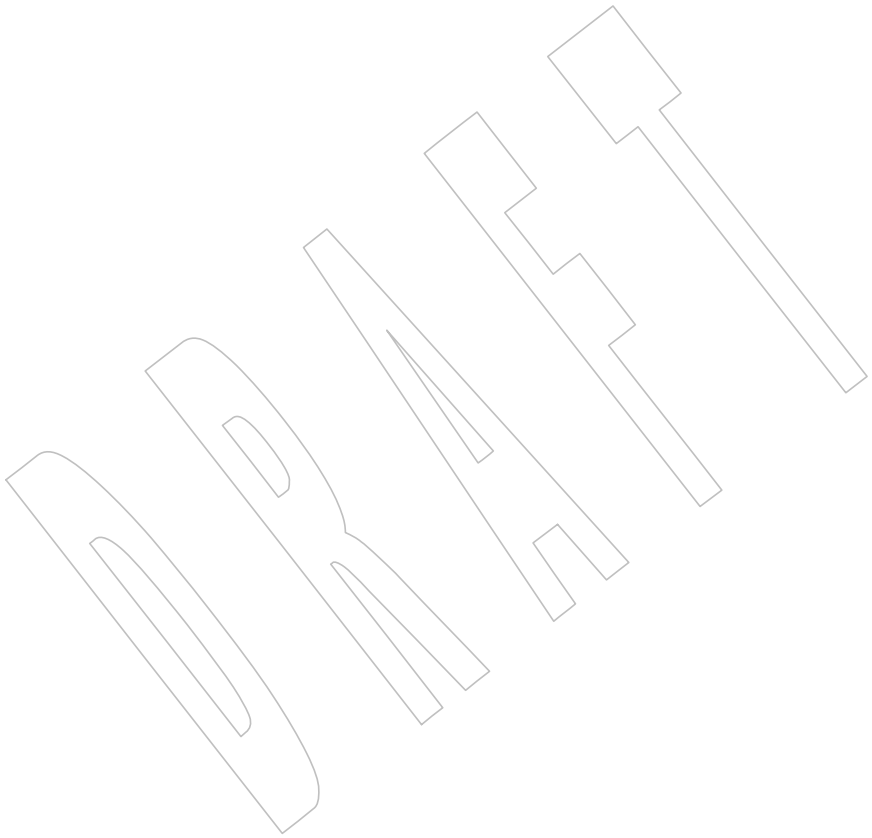
33749 **CHANGE HISTORY**

33750 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

33751 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

33752 **Issue 7**

33753 The *pthread_barrier_wait*( ) function is moved from the Barriers option to the Base.



NAME

pthread_barrierattr_destroy, pthread_barrierattr_init — destroy and initialize the barrier attributes object

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

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_barrierattr_getpshared( ), *pthread_barrierattr_setpshared*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**.

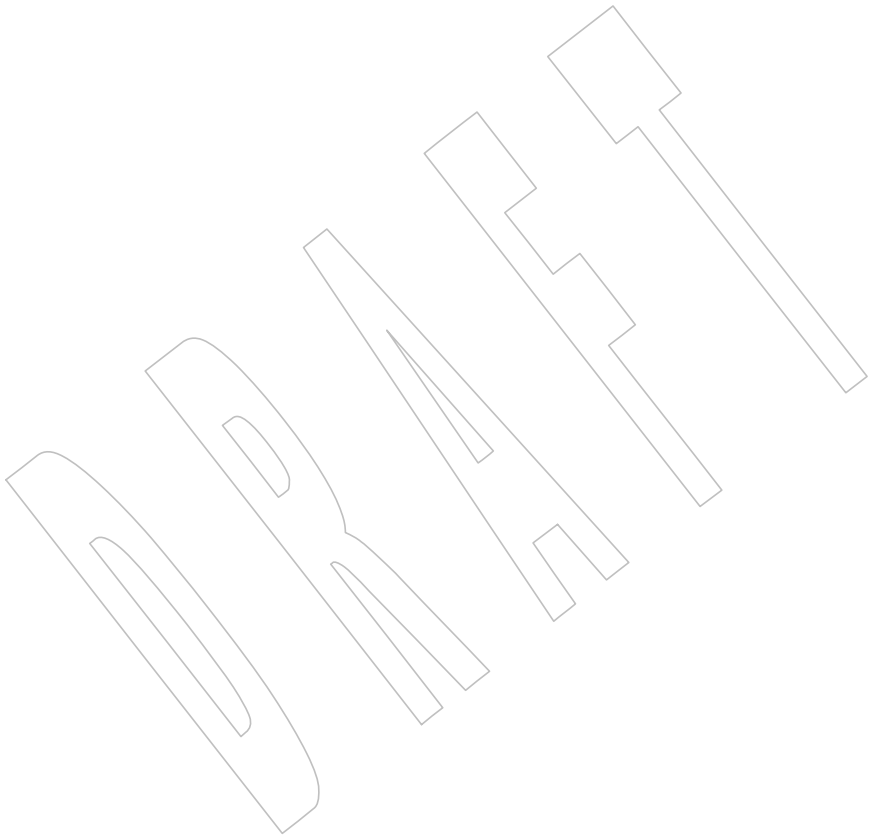
33793 **CHANGE HISTORY**

33794 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

33795 In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

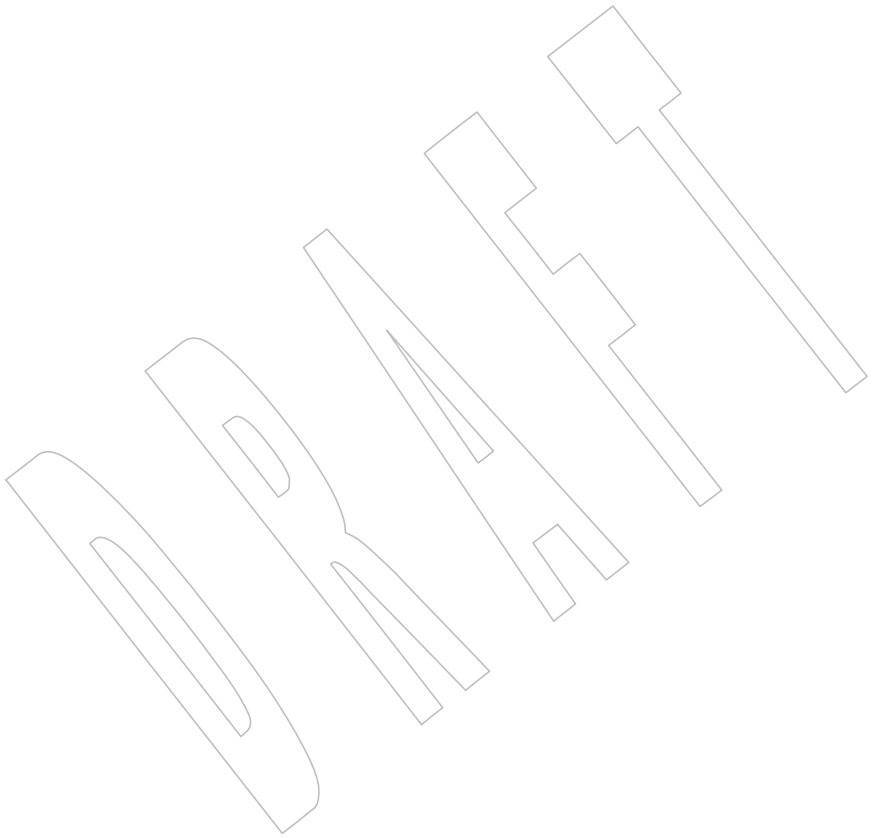
33796 **Issue 7**

33797 The *pthread_barrierattr_destroy*( ) and *pthread_barrierattr_init*( ) functions are moved from the
33798 Barriers option to the Base.



NAME

pthread_barrierattr_getpshared, pthread_barrierattr_setpshar



33835 **EXAMPLES**

33836 None.

33837 **APPLICATION USAGE**

33838 The *pthread_barrierattr_getpshared*( ) and *pthread_barrierattr_setpshared*( ) functions are part of the
33839 Thread Process-Shared Synchronization option and need not be provided on all
33840 implementations.

33841 **RATIONALE**

33842 None.

33843 **FUTURE DIRECTIONS**

33844 None.

33845 **SEE ALSO**

33846 *pthread_barrier_destroy*( ), *pthread_barrierattr_destroy*( ), *pthread_barrierattr_init*( ), the Base
33847 Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

33848 **CHANGE HISTORY**

33849 First released in Issue 6. Derived from IEEE Std 1003.1j-2000

33850 **Issue 7**

33851 The *pthread_barrierattr_getpshared*( ) and *pthread_barrierattr_setpshared*( ) functions are moved from
33852 the Barriers option.

33853 **NAME**

33854 pthread_barrierattr_init — initialize the barrier attributes object

33855 **SYNOPSIS**

33856
```
#include <pthread.h>
```

33857
```
int pthread_barrierattr_init(pthread_barrierattr_t *attr);
```

33858 **DESCRIPTION**

33859 Refer to *pthread_barrierattr_destroy( )*.

33860 **NAME**

33861 pthread_barrierattr_setpshared — set the process-shared attribute of the barrier attributes object

33862 **SYNOPSIS**

33863 TSH
```
#include <pthread.h>

int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr,
    int pshared);
```

33866 **DESCRIPTION**

33867 Refer to *pthread_barrierattr_getpshared( )*.

33868 **NAME**

33869 pthread_cancel — cancel execution of a thread

33870 **SYNOPSIS**

33871
```
#include <pthread.h>
```

33872
```
int pthread_cancel(pthread_t thread);
```

33873 **DESCRIPTION**

33874 The *pthread_cancel*( ) function shall request that *thread* be canceled. The target thread's
33875 cancelability state and type determines when the cancellation takes effect. When the cancellation
33876 is acted on, the cancellation cleanup handlers for *thread* shall be called. When the last
33877 cancellation cleanup handler returns, the thread-specific data destructor functions shall be called
33878 for *thread*. When the last destructor function returns, *thread* shall be terminated.

33879 The cancellation processing in the target thread shall run asynchronously with respect to the
33880 calling thread returning from *pthread_cancel*( ).

33881 **RETURN VALUE**

33882 If successful, the *pthread_cancel*( ) function shall return zero; otherwise, an error number shall be
33883 returned to indicate the error.

33884 **ERRORS**

33885 The *pthread_cancel*( ) function may fail if:

33886 [ESRCH] No thread could be found corresponding to that specified by the given thread
33887 ID.

33888 The *pthread_cancel*( ) function shall not return an error code of [EINTR].

33889 **EXAMPLES**

33890 None.

33891 **APPLICATION USAGE**

33892 None.

33893 **RATIONALE**

33894 Two alternative functions were considered for sending the cancellation notification to a thread.
33895 One would be to define a new SIGCANCEL signal that had the cancellation semantics when
33896 delivered; the other was to define the new *pthread_cancel*( ) function, which would trigger the
33897 cancellation semantics.

33898 The advantage of a new signal was that so much of the delivery criteria were identical to that
33899 used when trying to deliver a signal that making cancellation notification a signal was seen as
33900 consistent. Indeed, many implementations implement cancellation using a special signal. On the
33901 other hand, there would be no signal functions that could be used with this signal except
33902 *pthread_kill*( ), and the behavior of the delivered cancellation signal would be unlike any
33903 previously existing defined signal.

33904 The benefits of a special function include the recognition that this signal would be defined
33905 because of the similar delivery criteria and that this is the only common behavior between a
33906 cancellation request and a signal. In addition, the cancellation delivery mechanism does not
33907 have to be implemented as a signal. There are also strong, if not stronger, parallels with
33908 language exception mechanisms than with signals that are potentially obscured if the delivery
33909 mechanism is visibly closer to signals.

33910 In the end, it was considered that as there were so many exceptions to the use of the new signal
33911 with existing signals functions it would be misleading. A special function has resolved this
33912 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 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

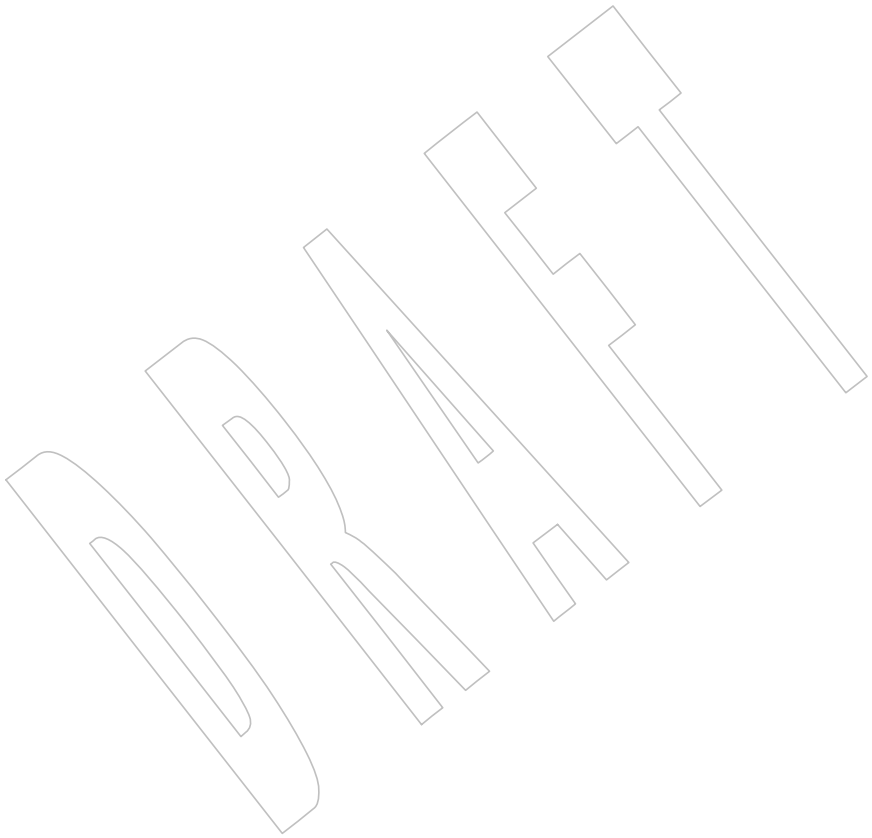
First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_cancel*( ) function is marked as part of the Threads option.

Issue 7

The *pthread_cancel*( ) function is moved from the Threads option to the Base.



33926 **NAME**

33927 pthread_cleanup_pop, pthread_cleanup_push — establish cancellation handlers

33928 **SYNOPSIS**

33929
```
#include <pthread.h>
```

33930
```
void pthread_cleanup_pop(int execute);
```
33931
```
void pthread_cleanup_push(void (*routine)(void*), void *arg);
```

33932 **DESCRIPTION**

33933 The *pthread_cleanup_pop*( ) function shall remove the routine at the top of the calling thread's
33934 cancellation cleanup stack and optionally invoke it (if *execute* is non-zero).

33935 The *pthread_cleanup_push*( ) function shall push the specified cancellation cleanup handler *routine*
33936 onto the calling thread's cancellation cleanup stack. The cancellation cleanup handler shall be
33937 popped from the cancellation cleanup stack and invoked with the argument *arg* when:

33938 • The thread exits (that is, calls *pthread_exit*( )).

33939 • The thread acts upon a cancellation request.

33940 • The thread calls *pthread_cleanup_pop*( ) with a non-zero *execute* argument.

33941 These functions may be implemented as macros. The application shall ensure that they appear
33942 as statements, and in pairs within the same lexical scope (that is, the *pthread_cleanup_push*( )
33943 macro may be thought to expand to a token list whose first token is `'{'` with
33944 *pthread_cleanup_pop*( ) expanding to a token list whose last token is the corresponding `'}'`).

33945 The effect of calling *longjmp*( ) or *siglongjmp*( ) is undefined if there have been any calls to
33946 *pthread_cleanup_push*( ) or *pthread_cleanup_pop*( ) made without the matching call since the jump
33947 buffer was filled. The effect of calling *longjmp*( ) or *siglongjmp*( ) from inside a cancellation
33948 cleanup handler is also undefined unless the jump buffer was also filled in the cancellation
33949 cleanup handler.

33950 The effect of the use of **return**, **break**, **continue**, and **goto** to prematurely leave a code block
33951 described by a pair of *pthread_cleanup_push*( ) and *pthread_cleanup_pop*( ) functions calls is
33952 undefined.

33953 **RETURN VALUE**

33954 The *pthread_cleanup_push*( ) and *pthread_cleanup_pop*( ) functions shall not return a value.

33955 **ERRORS**

33956 No errors are defined.

33957 These functions shall not return an error code of [EINTR].

33958 **EXAMPLES**

33959 The following is an example using thread primitives to implement a cancelable, writers-priority
33960 read-write lock:

33961
```
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;
```



```
33970       void
33971       waiting_reader_cleanup(void *arg)
33972       {
33973           rwlock *l;
33974           l = (rwlock *) arg;
33975           pthread_mutex_unlock(&l->lock);
33976       }
33977       void
33978       lock_for_read(rwlock *l)
33979       {
33980           pthread_mutex_lock(&l->lock);
33981           pthread_cleanup_push(waiting_reader_cleanup, l);
33982           while ((l->lock_count < 0) && (l->waiting_writers != 0))
33983               pthread_cond_wait(&l->rcond, &l->lock);
33984           l->lock_count++;
33985          /*
33986           * Note the pthread_cleanup_pop executes
33987           * waiting_reader_cleanup.
33988           */
33989           pthread_cleanup_pop(1);
33990       }
33991       void
33992       release_read_lock(rwlock *l)
33993       {
33994           pthread_mutex_lock(&l->lock);
33995           if (--l->lock_count == 0)
33996               pthread_cond_signal(&l->wcond);
33997           pthread_mutex_unlock(l);
33998       }
33999       void
34000       waiting_writer_cleanup(void *arg)
34001       {
34002           rwlock *l;
34003           l = (rwlock *) arg;
34004           if ((--l->waiting_writers == 0) && (l->lock_count >= 0)) {
34005              /*
34006               * This only happens if we have been canceled.
34007               */
34008               pthread_cond_broadcast(&l->wcond);
34009       }
34010           pthread_mutex_unlock(&l->lock);
34011       }
34012       void
34013       lock_for_write(rwlock *l)
34014       {
34015           pthread_mutex_lock(&l->lock);
34016           l->waiting_writers++;
34017           pthread_cleanup_push(waiting_writer_cleanup, l);
34018           while (l->lock_count != 0)
34019               pthread_cond_wait(&l->wcond, &l->lock);
34020           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 cancellation cleanup handlers, *pthread_cleanup_push*( ) and *pthread_cleanup_pop*( ), can be thought of as left and right parentheses. They always need to be matched.

34072 **RATIONALE**

34073 The restriction that the two routines that push and pop cancellation cleanup handlers,
34074 *pthread_cleanup_push*( ) and *pthread_cleanup_pop*( ), have to appear in the same lexical scope
34075 allows for efficient macro or compiler implementations and efficient storage management. A
34076 sample implementation of these routines as macros might look like this:

34077
```
#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;
```

34084
```
#define pthread_cleanup_pop(ex) \
    *__head = __cleanup_handler.next; \
    if (ex) (*__cleanup_handler.rtn)(__cleanup_handler.arg); \
}
```

34088 A more ambitious implementation of these routines might do even better by allowing the
34089 compiler to note that the cancellation cleanup handler is a constant and can be expanded inline.

34090 This volume of IEEE Std 1003.1-200x currently leaves unspecified the effect of calling *longjmp*( )
34091 from a signal handler executing in a POSIX System Interfaces function. If an implementation
34092 wants to allow this and give the programmer reasonable behavior, the *longjmp*( ) function has to
34093 call all cancellation cleanup handlers that have been pushed but not popped since the time
34094 *setjmp*( ) was called.

34095 Consider a multi-threaded function called by a thread that uses signals. If a signal were
34096 delivered to a signal handler during the operation of *qsort*( ) and that handler were to call
34097 *longjmp*( ) (which, in turn, did *not* call the cancellation cleanup handlers) the helper threads
34098 created by the *qsort*( ) function would not be canceled. Instead, they would continue to execute
34099 and write into the argument array even though the array might have been popped off the stack.

34100 Note that the specified cleanup handling mechanism is especially tied to the C language and,
34101 while the requirement for a uniform mechanism for expressing cleanup is language-
34102 independent, the mechanism used in other languages may be quite different. In addition, this
34103 mechanism is really only necessary due to the lack of a real exception mechanism in the C
34104 language, which would be the ideal solution.

34105 There is no notion of a cancellation cleanup-safe function. If an application has no cancellation
34106 points in its signal handlers, blocks any signal whose handler may have cancellation points
34107 while calling async-unsafe functions, or disables cancellation while calling async-unsafe
34108 functions, all functions may be safely called from cancellation cleanup routines.

34109 **FUTURE DIRECTIONS**

34110 None.

34111 **SEE ALSO**

34112 *pthread_cancel*( ), *pthread_setcancelstate*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
34113 **<pthread.h>**

34114 **CHANGE HISTORY**

34115 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

34116 **Issue 6**

34117 The *pthread_cleanup_pop*( ) and *pthread_cleanup_push*( ) functions are marked as part of the
34118 Threads option.

34119 The APPLICATION USAGE section is added.

The normative text is updated to avoid use of the term "must" for application requirements.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/88 is applied, updating the DESCRIPTION to describe the consequences of prematurely leaving a code block defined by the *pthread_cleanup_push*( ) and *pthread_cleanup_pop*( ) functions.

Issue 7

The *pthread_cleanup_pop*( ) and *pthread_cleanup_push*( ) functions are moved from the Threads option to the Base.

34127 **NAME**

34128 pthread_cond_broadcast, pthread_cond_signal — broadcast or signal a condition

34129 **SYNOPSIS**

34130
```
#include <pthread.h>
```

34131
```
int pthread_cond_broadcast(pthread_cond_t *cond);
```
34132
```
int pthread_cond_signal(pthread_cond_t *cond);
```

34133 **DESCRIPTION**

34134 These functions shall unblock threads blocked on a condition variable.

34135 The *pthread_cond_broadcast*( ) function shall unblock all threads currently blocked on the
34136 specified condition variable *cond*.

34137 The *pthread_cond_signal*( ) function shall unblock at least one of the threads that are blocked on
34138 the specified condition variable *cond* (if any threads are blocked on *cond*).

34139 If more than one thread is blocked on a condition variable, the scheduling policy shall determine
34140 the order in which threads are unblocked. When each thread unblocked as a result of a
34141 *pthread_cond_broadcast*( ) or *pthread_cond_signal*( ) returns from its call to *pthread_cond_wait*( ) or
34142 *pthread_cond_timedwait*( ), the thread shall own the mutex with which it called
34143 *pthread_cond_wait*( ) or *pthread_cond_timedwait*( ). The thread(s) that are unblocked shall contend
34144 for the mutex according to the scheduling policy (if applicable), and as if each had called
34145 *pthread_mutex_lock*( ).

34146 The *pthread_cond_broadcast*( ) or *pthread_cond_signal*( ) functions may be called by a thread
34147 whether or not it currently owns the mutex that threads calling *pthread_cond_wait*( ) or
34148 *pthread_cond_timedwait*( ) have associated with the condition variable during their waits;
34149 however, if predictable scheduling behavior is required, then that mutex shall be locked by the
34150 thread calling *pthread_cond_broadcast*( ) or *pthread_cond_signal*( ).

34151 The *pthread_cond_broadcast*( ) and *pthread_cond_signal*( ) functions shall have no effect if there are
34152 no threads currently blocked on *cond*.

34153 **RETURN VALUE**

34154 If successful, the *pthread_cond_broadcast*( ) and *pthread_cond_signal*( ) functions shall return zero;
34155 otherwise, an error number shall be returned to indicate the error.

34156 **ERRORS**

34157 The *pthread_cond_broadcast*( ) and *pthread_cond_signal*( ) function may fail if:

34158 [EINVAL] The value *cond* does not refer to an initialized condition variable.

34159 These functions shall not return an error code of [EINTR].

34160 **EXAMPLES**

34161 None.

34162 **APPLICATION USAGE**

34163 The *pthread_cond_broadcast*( ) function is used whenever the shared-variable state has been
34164 changed in a way that more than one thread can proceed with its task. Consider a single
34165 producer/multiple consumer problem, where the producer can insert multiple items on a list
34166 that is accessed one item at a time by the consumers. By calling the *pthread_cond_broadcast*( )
34167 function, the producer would notify all consumers that might be waiting, and thereby the
34168 application would receive more throughput on a multi-processor. In addition,
34169 *pthread_cond_broadcast*( ) makes it easier to implement a read-write lock. The
34170 *pthread_cond_broadcast*( ) function is needed in order to wake up all waiting readers when a
34171 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 1003.1-200x 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 1003.1-200x, Section 4.10, Memory Synchronization, **<pthread.h>**

CHANGE HISTORY

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

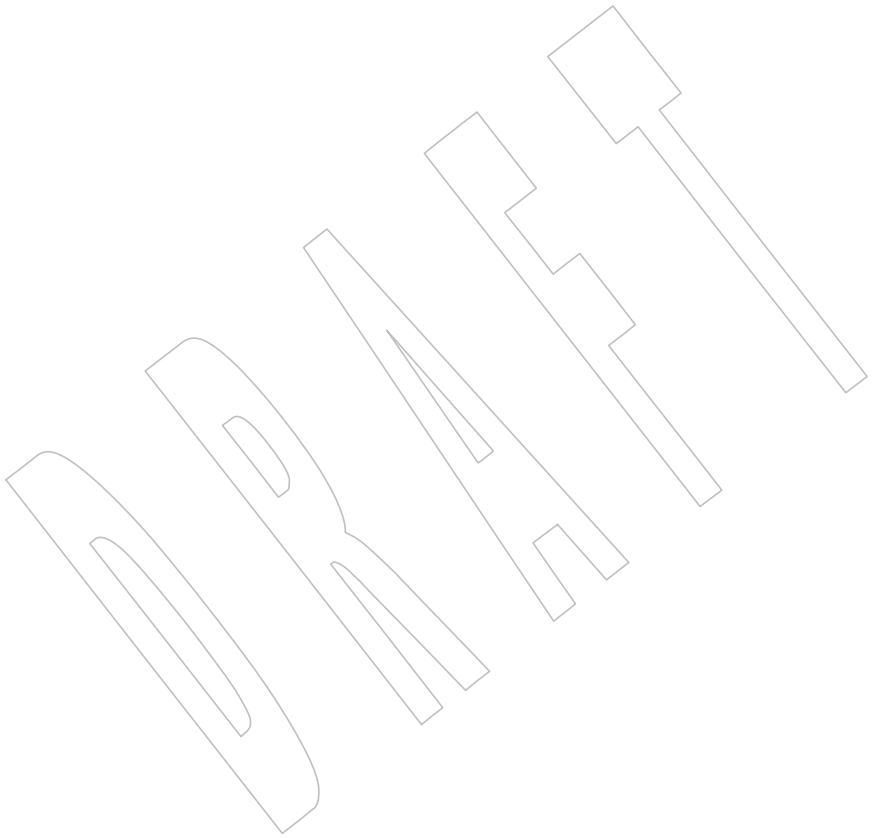
Issue 6

The *pthread_cond_broadcast*( ) and *pthread_cond_signal*( ) functions are marked as part of the Threads option.

The APPLICATION USAGE section is added.

Issue 7

The *pthread_cond_broadcast*( ) and *pthread_cond_signal*( ) functions are moved from the Threads option to the Base.



NAME

pthread_cond_destroy, pthread_cond_init — destroy and initialize condition variables

SYNOPSIS

```
#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.

[ENOMEM] Insufficient memory exists to initialize the condition variable.

The *pthread_cond_init*( ) function may fail if:

[EBUSY] 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 [EINTR].

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:

```
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(&rp->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 1003.1-200x, **<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.

Issue 7

The *pthread_cond_destroy*( ) and *pthread_cond_init*( ) functions are moved from the Threads option to the Base.

pthread_cond_signal( )

34346 **NAME**

34347 pthread_cond_signal — signal a condition

34348 **SYNOPSIS**

34349 `#include <pthread.h>`

34350 `int pthread_cond_signal(pthread_cond_t *cond);`

34351 **DESCRIPTION**

34352 Refer to *pthread_cond_broadcast( )*.

34353 **NAME**

34354 pthread_cond_timedwait, pthread_cond_wait — wait on a condition

34355 **SYNOPSIS**

34356
```
#include <pthread.h>
```

34357
```
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);
```

34362 **DESCRIPTION**

34363 The *pthread_cond_timedwait*( ) and *pthread_cond_wait*( ) functions shall block on a condition
34364 variable. They shall be called with *mutex* locked by the calling thread or undefined behavior
34365 results.

34366 These functions atomically release *mutex* and cause the calling thread to block on the condition
34367 variable *cond*; atomically here means "atomically with respect to access by another thread to the
34368 mutex and then the condition variable". That is, if another thread is able to acquire the mutex
34369 after the about-to-block thread has released it, then a subsequent call to *pthread_cond_broadcast*( )
34370 or *pthread_cond_signal*( ) in that thread shall behave as if it were issued after the about-to-block
34371 thread has blocked.

34372 Upon successful return, the mutex shall have been locked and shall be owned by the calling
34373 thread. If *mutex* is a robust mutex where an owner terminated while holding the lock and the
34374 state is recoverable, the mutex shall be acquired even though the function returns an error code.

34375 When using condition variables there is always a Boolean predicate involving shared variables
34376 associated with each condition wait that is true if the thread should proceed. Spurious wakeups
34377 from the *pthread_cond_timedwait*( ) or *pthread_cond_wait*( ) functions may occur. Since the return
34378 from *pthread_cond_timedwait*( ) or *pthread_cond_wait*( ) does not imply anything about the value of
34379 this predicate, the predicate should be re-evaluated upon such return.

34380 When a thread waits on a condition variable, having specified a particular mutex to either the
34381 *pthread_cond_timedwait*( ) or the *pthread_cond_wait*( ) operation, a dynamic binding is formed
34382 between that mutex and condition variable that remains in effect as long as at least one thread is
34383 blocked on the condition variable. During this time, the effect of an attempt by any thread to
34384 wait on that condition variable using a different mutex is undefined. Once all waiting threads
34385 have been unblocked (as by the *pthread_cond_broadcast*( ) operation), the next wait operation on
34386 that condition variable shall form a new dynamic binding with the mutex specified by that wait
34387 operation. Even though the dynamic binding between condition variable and mutex may be
34388 removed or replaced between the time a thread is unblocked from a wait on the condition
34389 variable and the time that it returns to the caller or begins cancellation cleanup, the unblocked
34390 thread shall always re-acquire the mutex specified in the condition wait operation call from
34391 which it is returning.

34392 A condition wait (whether timed or not) is a cancellation point. When the cancelability type of a
34393 thread is set to PTHREAD_CANCEL_DEFERRED, a side effect of acting upon a cancellation
34394 request while in a condition wait is that the mutex is (in effect) re-acquired before calling the first
34395 cancellation cleanup handler. The effect is as if the thread were unblocked, allowed to execute up
34396 to the point of returning from the call to *pthread_cond_timedwait*( ) or *pthread_cond_wait*( ), but at
34397 that point notices the cancellation request and instead of returning to the caller of
34398 *pthread_cond_timedwait*( ) or *pthread_cond_wait*( ), starts the thread cancellation activities, which
34399 includes calling cancellation 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.

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 cancellation 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.

[EINVAL] The *abstime* argument specified a nanosecond value less than zero or greater than or equal to 1000 million.

[ENOTRECOVERABLE]
The state protected by the mutex is not recoverable. The mutex is not locked.

[EOWNERDEAD]
The mutex is a robust mutex and the process containing the previous owner thread terminated while holding the mutex lock. The mutex lock has been acquired and it is up to the new owner to make the state consistent.

These functions may fail if:

[EINVAL] The value specified by *cond* or *mutex* is invalid.

[EOWNERDEAD]
The mutex is a robust mutex and the previous owning thread terminated while holding the mutex lock. The mutex lock has been acquired and it is up to the new owner to make the state consistent.

[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].

34441 **EXAMPLES**

34442 None.

34443 **APPLICATION USAGE**

34444 Applications that have assumed that non-zero return values are errors will need updating for
34445 use with robust mutexes, since a valid return for a thread acquiring a mutex which is protecting
34446 a currently inconsistent state is [EOWNERDEAD]. Applications that do not check the error
34447 returns, due to ruling out the possibility of such errors arising, should not use robust mutexes. If
34448 an application is supposed to work with normal and robust mutexes, it should check all return
34449 values for error conditions and if necessary take appropriate action.

34450 **RATIONALE**

34451 **Condition Wait Semantics**

34452 It is important to note that when *pthread_cond_wait*( ) and *pthread_cond_timedwait*( ) return
34453 without error, the associated predicate may still be false. Similarly, when
34454 *pthread_cond_timedwait*( ) returns with the timeout error, the associated predicate may be true
34455 due to an unavoidable race between the expiration of the timeout and the predicate state change.

34456 The application needs to recheck the predicate on any return because it cannot be sure there is
34457 another thread waiting on the thread to handle the signal, and if there is not then the signal is
34458 lost. The burden is on the application to check the predicate.

34459 Some implementations, particularly on a multi-processor, may sometimes cause multiple
34460 threads to wake up when the condition variable is signaled simultaneously on different
34461 processors.

34462 In general, whenever a condition wait returns, the thread has to re-evaluate the predicate
34463 associated with the condition wait to determine whether it can safely proceed, should wait
34464 again, or should declare a timeout. A return from the wait does not imply that the associated
34465 predicate is either true or false.

34466 It is thus recommended that a condition wait be enclosed in the equivalent of a "while loop"
34467 that checks the predicate.

34468 **Timed Wait Semantics**

34469 An absolute time measure was chosen for specifying the timeout parameter for two reasons.
34470 First, a relative time measure can be easily implemented on top of a function that specifies
34471 absolute time, but there is a race condition associated with specifying an absolute timeout on top
34472 of a function that specifies relative timeouts. For example, assume that *clock_gettime*( ) returns
34473 the current time and *cond_relative_timed_wait*( ) uses relative timeouts:

34474 ```
clock_gettime(CLOCK_REALTIME, &now)
34475 reltime = sleep_til_this_absolute_time -now;
34476 cond_relative_timed_wait(c, m, &reltime);
```

34477 If the thread is preempted between the first statement and the last statement, the thread blocks
34478 for too long. Blocking, however, is irrelevant if an absolute timeout is used. An absolute timeout
34479 also need not be recomputed if it is used multiple times in a loop, such as that enclosing a
34480 condition wait.

34481 For cases when the system clock is advanced discontinuously by an operator, it is expected that
34482 implementations process any timed wait expiring at an intervening time as if that time had
34483 actually occurred.

  
```

34484 **Cancellation and Condition Wait**

34485 A condition wait, whether timed or not, is a cancellation point. That is, the functions
34486 *pthread_cond_wait*( ) or *pthread_cond_timedwait*( ) are points where a pending (or concurrent)
34487 cancellation request is noticed. The reason for this is that an indefinite wait is possible at these
34488 points—whatever event is being waited for, even if the program is totally correct, might never
34489 occur; for example, some input data being awaited might never be sent. By making condition
34490 wait a cancellation point, the thread can be canceled and perform its cancellation cleanup
34491 handler even though it may be stuck in some indefinite wait.

34492 A side effect of acting on a cancellation request while a thread is blocked on a condition variable
34493 is to re-acquire the mutex before calling any of the cancellation cleanup handlers. This is done in
34494 order to ensure that the cancellation cleanup handler is executed in the same state as the critical
34495 code that lies both before and after the call to the condition wait function. This rule is also
34496 required when interfacing to POSIX threads from languages, such as Ada or C++, which may
34497 choose to map cancellation onto a language exception; this rule ensures that each exception
34498 handler guarding a critical section can always safely depend upon the fact that the associated
34499 mutex has already been locked regardless of exactly where within the critical section the
34500 exception was raised. Without this rule, there would not be a uniform rule that exception
34501 handlers could follow regarding the lock, and so coding would become very cumbersome.

34502 Therefore, since *some* statement has to be made regarding the state of the lock when a
34503 cancellation is delivered during a wait, a definition has been chosen that makes application
34504 coding most convenient and error free.

34505 When acting on a cancellation request while a thread is blocked on a condition variable, the
34506 implementation is required to ensure that the thread does not consume any condition signals
34507 directed at that condition variable if there are any other threads waiting on that condition
34508 variable. This rule is specified in order to avoid deadlock conditions that could occur if these
34509 two independent requests (one acting on a thread and the other acting on the condition variable)
34510 were not processed independently.

34511 **Performance of Mutexes and Condition Variables**

34512 Mutexes are expected to be locked only for a few instructions. This practice is almost
34513 automatically enforced by the desire of programmers to avoid long serial regions of execution
34514 (which would reduce total effective parallelism).

34515 When using mutexes and condition variables, one tries to ensure that the usual case is to lock the
34516 mutex, access shared data, and unlock the mutex. Waiting on a condition variable should be a
34517 relatively rare situation. For example, when implementing a read-write lock, code that acquires a
34518 read-lock typically needs only to increment the count of readers (under mutual-exclusion) and
34519 return. The calling thread would actually wait on the condition variable only when there is
34520 already an active writer. So the efficiency of a synchronization operation is bounded by the cost
34521 of mutex lock/unlock and not by condition wait. Note that in the usual case there is no context
34522 switch.

34523 This is not to say that the efficiency of condition waiting is unimportant. Since there needs to be
34524 at least one context switch per Ada rendezvous, the efficiency of waiting on a condition variable
34525 is important. The cost of waiting on a condition variable should be little more than the minimal
34526 cost for a context switch plus the time to unlock and lock the mutex.

Features of Mutexes and Condition Variables

It had been suggested that the mutex acquisition and release be decoupled from condition wait. This was rejected because it is the combined nature of the operation that, in fact, facilitates realtime implementations. Those implementations can atomically move a high-priority thread between the condition variable and the mutex in a manner that is transparent to the caller. This can prevent extra context switches and provide more deterministic acquisition of a mutex when the waiting thread is signaled. Thus, fairness and priority issues can be dealt with directly by the scheduling discipline. Furthermore, the current condition wait operation matches existing practice.

Scheduling Behavior of Mutexes and Condition Variables

Synchronization primitives that attempt to interfere with scheduling policy by specifying an ordering rule are considered undesirable. Threads waiting on mutexes and condition variables are selected to proceed in an order dependent upon the scheduling policy rather than in some fixed order (for example, FIFO or priority). Thus, the scheduling policy determines which thread(s) are awakened and allowed to proceed.

Timed Condition Wait

The *pthread_cond_timedwait*( ) function allows an application to give up waiting for a particular condition after a given amount of time. An example of its use follows:

```
(void) pthread_mutex_lock(&t.mn);
        t.waiters++;
    clock_gettime(CLOCK_REALTIME, &ts);
    ts.tv_sec += 5;
    rc = 0;
    while (! mypredicate(&t) && rc == 0)
        rc = pthread_cond_timedwait(&t.cond, &t.mn, &ts);
    t.waiters--;
    if (rc == 0) setmystate(&t);
(void) pthread_mutex_unlock(&t.mn);
```

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 1003.1-200x, Section 4.10, Memory Synchronization, **<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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/89 is applied, updating the DESCRIPTION for consistency with the *pthread_cond_destroy*( ) function that states it is safe to destroy an initialized condition variable upon which no threads are currently blocked.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/90 is applied, updating words in the DESCRIPTION from "the cancelability enable state" to "the cancelability type".

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/91 is applied, updating the ERRORS section to remove the error case related to *abstime* from the *pthread_cond_wait*( ) function, and to make the error case related to *abstime* mandatory for *pthread_cond_timedwait*( ) for consistency with other functions.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/92 is applied, adding a new paragraph to the RATIONALE section stating that an application should check the predicate on any return from this function.

Issue 7

SD5-XSH-ERN-44 is applied, changing the definition of the "shall fail" case of the [EINVAL] error.

Changes are made from The Open Group Technical Standard, 2006, Extended API Set Part 3.

The *pthread_cond_timedwait*( ) and *pthread_cond_wait*( ) functions are moved from the Threads option to the Base.

NAME

pthread_condattr_destroy, pthread_condattr_init — destroy and initialize the condition variable attributes object

SYNOPSIS

```
#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, anyP0C85.nitiaf178.35 Td2.014 T0r a cond99UR40cau1ENTd(ec1e.24n7.4su) indefined ifond2.54 Tw[oy

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 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

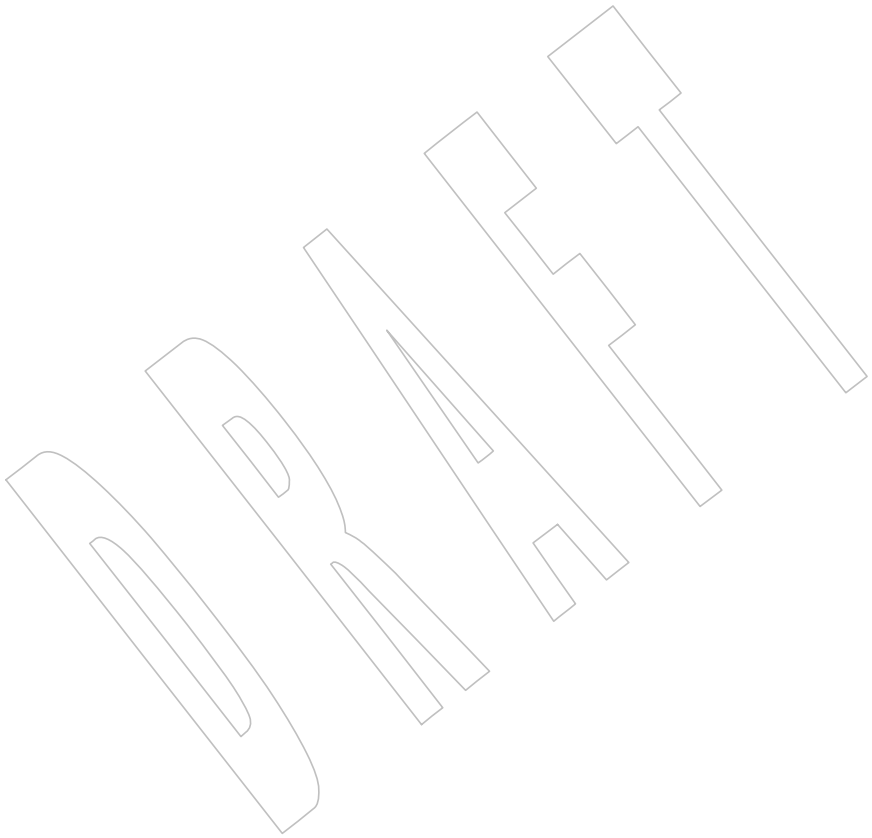
First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_condattr_destroy*( ) and *pthread_condattr_init*( ) functions are marked as part of the Threads option.

Issue 7

The *pthread_condattr_destroy*( ) and *pthread_condattr_init*( ) functions are moved from the Threads option to the Base.



34650 **NAME**

34651 pthread_condattr_getclock, pthread_condattr_setclock — get and set the clock selection
34652 condition variable attribute

34653 **SYNOPSIS**

34654
```
#include <pthread.h>
```

34655
```
int pthread_condattr_getclock(const pthread_condattr_t *restrict attr,
    clockid_t *restrict clock_id);
```
34656
34657
```
int pthread_condattr_setclock(pthread_condattr_t *attr,
    clockid_t clock_id);
```
34658

34659 **DESCRIPTION**

34660 The *pthread_condattr_getclock*( ) function shall obtain the value of the *clock* attribute from the
34661 attributes object referenced by *attr*. The *pthread_condattr_setclock*( ) function shall set the *clock*
34662 attribute in an initialized attributes object referenced by *attr*. If *pthread_condattr_setclock*( ) is
34663 called with a *clock_id* argument that refers to a CPU-time clock, the call shall fail.

34664 The *clock* attribute is the clock ID of the clock that shall be used to measure the timeout service of
34665 *pthread_cond_timedwait*( ). The default value of the *clock* attribute shall refer to the system clock.

34666 **RETURN VALUE**

34667 If successful, the *pthread_condattr_getclock*( ) function shall return zero and store the value of the
34668 clock attribute of *attr* into the object referenced by the *clock_id* argument. Otherwise, an error
34669 number shall be returned to indicate the error.

34670 If successful, the *pthread_condattr_setclock*( ) function shall return zero; otherwise, an error
34671 number shall be returned to indicate the error.

34672 **ERRORS**

34673 These functions may fail if:

34674 [EINVAL] The value specified by *attr* is invalid.

34675 The *pthread_condattr_setclock*( ) function may fail if:

34676 [EINVAL] The value specified by *clock_id* does not refer to a known clock, or is a CPU-
34677 time clock.

34678 These functions shall not return an error code of [EINTR].

34679 **EXAMPLES**

34680 None.

34681 **APPLICATION USAGE**

34682 None.

34683 **RATIONALE**

34684 None.

34685 **FUTURE DIRECTIONS**

34686 None.

34687 **SEE ALSO**

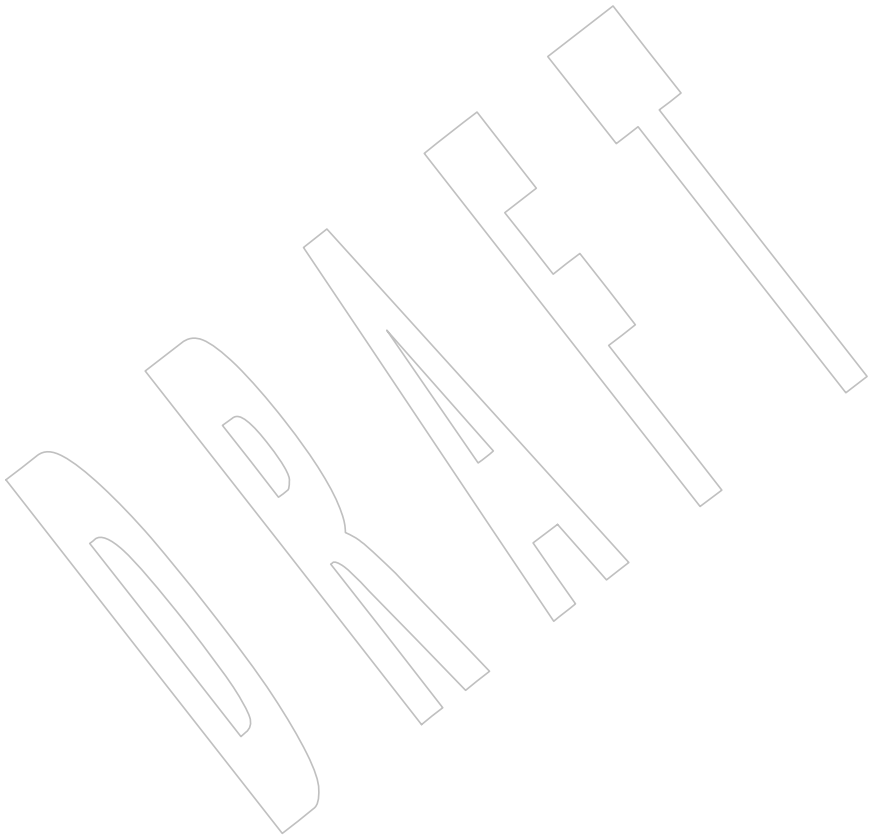
34688 *pthread_cond_destroy*( ), *pthread_cond_timedwait*( ), *pthread_condattr_destroy*( ),
34689 *pthread_condattr_getpshared*( ) (on page 1107),1 *pthread_condattr_init*( ),
34690 *pthread_condattr_setpshared*( ) (on page 1111),1 *pthread_create*( ), *pthread_mutex_init*( ), the Base
34691 Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

34692 **CHANGE HISTORY**

34693 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

34694 **Issue 7**

34695 The *pthread_condattr_getclock*( ) and *pthread_condattr_setclock*( ) functions are moved from the
34696 Clock Selection option to the Base.



34697 **NAME**

34698 pthread_condattr_getpshared, pthread_condattr_setpshared — get and set the process-shared
34699 condition variable attributes

34700 **SYNOPSIS**

34701 TSH
```
#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);
```

34706 **DESCRIPTION**

34707 The *pthread_condattr_getpshared*( ) function shall obtain the value of the *process-shared* attribute
34708 from the attributes object referenced by *attr*. The *pthread_condattr_setpshared*( ) function shall set
34709 the *process-shared* attribute in an initialized attributes object referenced by *attr*.

34710 The *process-shared* attribute is set to PTHREAD_PROCESS_SHARED to permit a condition
34711 variable to be operated upon by any thread that has access to the memory where the condition
34712 variable is allocated, even if the condition variable is allocated in memory that is shared by
34713 multiple processes. If the *process-shared* attribute is PTHREAD_PROCESS_PRIVATE, the
34714 condition variable shall only be operated upon by threads created within the same process as the
34715 thread that initialized the condition variable; if threads of differing processes attempt to operate
34716 on such a condition variable, the behavior is undefined. The default value of the attribute is
34717 PTHREAD_PROCESS_PRIVATE.

34718 **RETURN VALUE**

34719 If successful, the *pthread_condattr_setpshared*( ) function shall return zero; otherwise, an error
34720 number shall be returned to indicate the error.

34721 If successful, the *pthread_condattr_getpshared*( ) function shall return zero and store the value of
34722 the *process-shared* attribute of *attr* into the object referenced by the *pshared* parameter. Otherwise,
34723 an error number shall be returned to indicate the error.

34724 **ERRORS**

34725 The *pthread_condattr_getpshared*( ) and *pthread_condattr_setpshared*( ) functions may fail if:

34726 [EINVAL] The value specified by *attr* is invalid.

34727 The *pthread_condattr_setpshared*( ) function may fail if:

34728 [EINVAL] The new value specified for the attribute is outside the range of legal values
34729 for that attribute.

34730 These functions shall not return an error code of [EINTR].

34731 **EXAMPLES**

34732 None.

34733 **APPLICATION USAGE**

34734 None.

34735 **RATIONALE**

34736 None.

34737 **FUTURE DIRECTIONS**

34738 None.

34739 **SEE ALSO**

34740 *pthread_create*( ), *pthread_cond_destroy*( ), *pthread_condattr_destroy*( ), *pthread_mutex_destroy*( ), the
34741 Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

34742 **CHANGE HISTORY**

34743 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

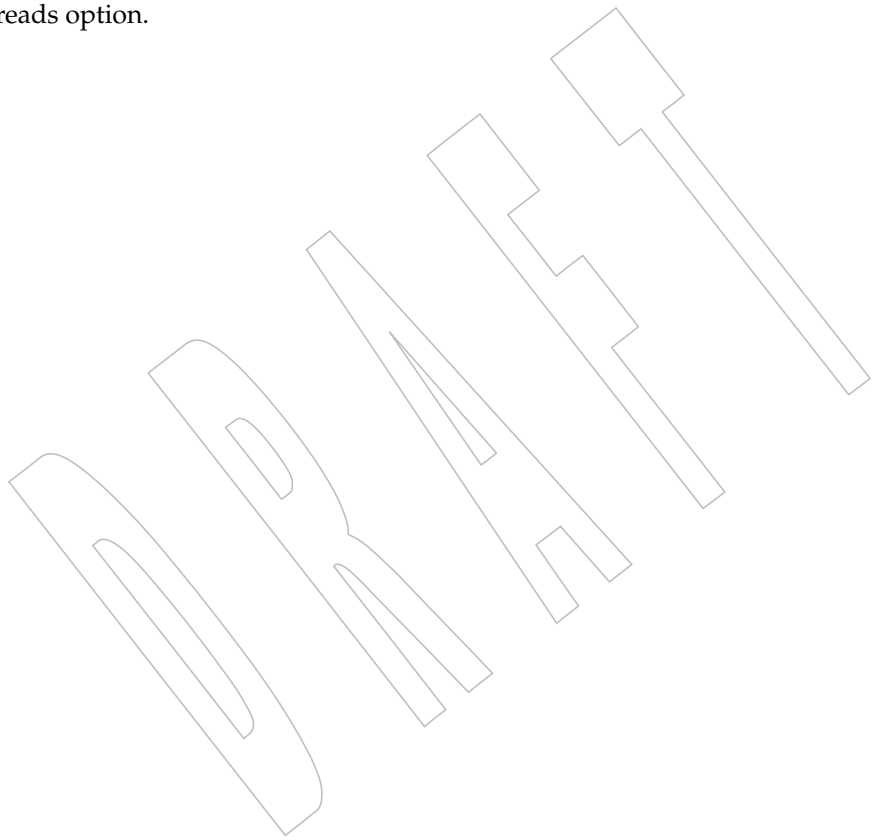
34744 **Issue 6**

34745 The *pthread_condattr_getpshared*( ) and *pthread_condattr_setpshared*( ) functions are marked as part
34746 of the Threads and Thread Process-Shared Synchronization options.

34747 The **restrict** keyword is added to the *pthread_condattr_getpshared*( ) prototype for alignment with
34748 the ISO/IEC 9899:1999 standard.

34749 **Issue 7**

34750 The *pthread_condattr_getpshared*( ) and *pthread_condattr_setpshared*( ) functions are moved from the
34751 Threads option.



34752 **NAME**

34753 pthread_condattr_init — initialize the condition variable attributes object

34754 **SYNOPSIS**

34755
```
#include <pthread.h>
```

34756
```
int pthread_condattr_init(pthread_condattr_t *attr);
```

34757 **DESCRIPTION**

34758 Refer to *pthread_condattr_destroy( )*.

pthread_condattr_setclock( )

34759 **NAME**

34760 pthread_condattr_setclock — set the clock selection condition variable attribute

34761 **SYNOPSIS**

34762
```
#include <pthread.h>
```

34763
```
int pthread_condattr_setclock(pthread_condattr_t *attr,
```
34764
```
    clockid_t clock_id);
```

34765 **DESCRIPTION**

34766 Refer to *pthread_condattr_getclock( )*.

34767 **NAME**

34768 pthread_condattr_setpshared — set the process-shared condition variable attribute

34769 **SYNOPSIS**

34770 TSH
```
#include <pthread.h>

int pthread_condattr_setpshared(pthread_condattr_t *attr,
    int pshared);
```

34773 **DESCRIPTION**

34774 Refer to *pthread_condattr_getpshared( )*.

34775 **NAME**

34776 pthread_create — thread creation

34777 **SYNOPSIS**

34778
```
#include <pthread.h>
```

34779
```
int pthread_create(pthread_t *restrict thread,
    const pthread_attr_t *restrict attr,
    void *(*start_routine)(void*), void *restrict arg);
```

34782 **DESCRIPTION**

34783 The *pthread_create*( ) function shall create a new thread, with attributes specified by *attr*, within a
34784 process. If *attr* is NULL, the default attributes shall be used. If the attributes specified by *attr* are
34785 modified later, the thread's attributes shall not be affected. Upon successful completion,
34786 *pthread_create*( ) shall store the ID of the created thread in the location referenced by *thread*.

34787 The thread is created executing *start_routine* with *arg* as its sole argument. If the *start_routine*
34788 returns, the effect shall be as if there was an implicit call to *pthread_exit*( ) using the return value
34789 of *start_routine* as the exit status. Note that the thread in which *main*( ) was originally invoked
34790 differs from this. When it returns from *main*( ), the effect shall be as if there was an implicit call to
34791 *exit*( ) using the return value of *main*( ) as the exit status.

34792 The signal state of the new thread shall be initialized as follows:

34793 • The signal mask shall be inherited from the creating thread.

34794 • The set of signals pending for the new thread shall be empty.

34795 XSI The alternate stack shall not be inherited.

34796 The floating-point environment shall be inherited from the creating thread.

34797 If *pthread_create*( ) fails, no new thread is created and the contents of the location referenced by
34798 *thread* are undefined.

34799 TCT If _POSIX_THREAD_CPUTIME is defined, the new thread shall have a CPU-time clock
34800 accessible, and the initial value of this clock shall be set to zero.

34801 **RETURN VALUE**

34802 If successful, the *pthread_create*( ) function shall return zero; otherwise, an error number shall be
34803 returned to indicate the error.

34804 **ERRORS**

34805 The *pthread_create*( ) function shall fail if:

34806 [EAGAIN] The system lacked the necessary resources to create another thread, or the
34807 system-imposed limit on the total number of threads in a process
34808 {PTHREAD_THREADS_MAX} would be exceeded.

34809 [EPERM] The caller does not have appropriate permission to set the required scheduling
34810 parameters or scheduling policy.

34811 The *pthread_create*( ) function may fail if:

34812 [EINVAL] The attributes specified by *attr* are invalid.

34813 The *pthread_create*( ) function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

There is no requirement on the implementation that the ID of the created thread be available before the newly created thread starts executing. The calling thread can obtain the ID of the created thread through the return value of the *pthread_create*( ) function, and the newly created thread can obtain its ID by a call to *pthread_self*( ).

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

- 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 1003.1-200x, Section 4.10, Memory Synchronization, **<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.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/44 is applied, adding text that the alternate stack is not inherited.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/93 is applied, updating the ERRORS section to remove the mandatory [EINVAL] error (“The value specified by *attr* is invalid”), and adding the optional [EINVAL] error (“The attributes specified by *attr* are invalid”).

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/94 is applied, adding the APPLICATION USAGE section.

Issue 7

The *pthread_create*( ) function is moved from the Threads option to the Base.

34893 **NAME**

34894 pthread_detach — detach a thread

34895 **SYNOPSIS**

34896
```
#include <pthread.h>
```

34897
```
int pthread_detach(pthread_t thread);
```

34898 **DESCRIPTION**

34899 The *pthread_detach*( ) function shall indicate to the implementation that storage for the thread
34900 *thread* can be reclaimed when that thread terminates. If *thread* has not terminated,
34901 *pthread_detach*( ) shall not cause it to terminate. The effect of multiple *pthread_detach*( ) calls on the
34902 same target thread is unspecified.

34903 **RETURN VALUE**

34904 If the call succeeds, *pthread_detach*( ) shall return 0; otherwise, an error number shall be returned
34905 to indicate the error.

34906 **ERRORS**

34907 The *pthread_detach*( ) function may fail if:

34908 [EINVAL] The implementation has detected that the value specified by *thread* does not
34909 refer to a joinable thread.

34910 [ESRCH] No thread could be found corresponding to that specified by the given thread
34911 ID.

34912 The *pthread_detach*( ) function shall not return an error code of [EINTR].

34913 **EXAMPLES**

34914 None.

34915 **APPLICATION USAGE**

34916 None.

34917 **RATIONALE**

34918 The *pthread_join*( ) or *pthread_detach*( ) functions should eventually be called for every thread that
34919 is created so that storage associated with the thread may be reclaimed.

34920 It has been suggested that a "detach" function is not necessary; the *detachstate* thread creation
34921 attribute is sufficient, since a thread need never be dynamically detached. However, need arises
34922 in at least two cases:

34923 1. In a cancellation handler for a *pthread_join*( ) it is nearly essential to have a
34924 *pthread_detach*( ) function in order to detach the thread on which *pthread_join*( ) was
34925 waiting. Without it, it would be necessary to have the handler do another *pthread_join*( ) to
34926 attempt to detach the thread, which would both delay the cancellation processing for an
34927 unbounded period and introduce a new call to *pthread_join*( ), which might itself need a
34928 cancellation handler. A dynamic detach is nearly essential in this case.

34929 2. In order to detach the "initial thread" (as may be desirable in processes that set up server
34930 threads).

34931 **FUTURE DIRECTIONS**

34932 None.

34933 **SEE ALSO**

34934 *pthread_join( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

34935 **CHANGE HISTORY**

34936 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

34937 **Issue 6**

34938 The *pthread_detach*( ) function is marked as part of the Threads option.

34939 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/95 is applied, updating the ERRORS
34940 section so that the [EINVAL] and [ESRCH] error cases become optional.

34941 **Issue 7**

34942 The *pthread_detach*( ) function is moved from the Threads option to the Base.

34943 **NAME**

34944 pthread_equal — compare thread IDs

34945 **SYNOPSIS**

34946
```
#include <pthread.h>
```

34947
```
int pthread_equal(pthread_t t1, pthread_t t2);
```

34948 **DESCRIPTION**

34949 This function shall compare the thread IDs *t1* and *t2*.

34950 **RETURN VALUE**

34951 The *pthread_equal*( ) function shall return a non-zero value if *t1* and *t2* are equal; otherwise, zero
34952 shall be returned.

34953 If either *t1* or *t2* are not valid thread IDs, the behavior is undefined.

34954 **ERRORS**

34955 No errors are defined.

34956 The *pthread_equal*( ) function shall not return an error code of [EINTR].

34957 **EXAMPLES**

34958 None.

34959 **APPLICATION USAGE**

34960 None.

34961 **RATIONALE**

34962 Implementations may choose to define a thread ID as a structure. This allows additional
34963 flexibility and robustness over using an **int**. For example, a thread ID could include a sequence
34964 number that allows detection of "dangling IDs" (copies of a thread ID that has been detached).
34965 Since the C language does not support comparison on structure types, the *pthread_equal*( )
34966 function is provided to compare thread IDs.

34967 **FUTURE DIRECTIONS**

34968 None.

34969 **SEE ALSO**

34970 *pthread_create*( ), *pthread_self*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

34971 **CHANGE HISTORY**

34972 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

34973 **Issue 6**

34974 The *pthread_equal*( ) function is marked as part of the Threads option.

34975 **Issue 7**

34976 The *pthread_equal*( ) function is moved from the Threads option to the Base.

NAME

pthread_exit — thread termination

SYNOPSIS

```
#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 cancellation cleanup handlers that have been pushed and not yet popped shall be popped in the reverse order that they were pushed and then executed. After all cancellation cleanup handlers have been executed, if the thread has any thread-specific data, appropriate destructor functions shall be called in an unspecified order. Thread termination does not release any application visible process resources, including, but not limited to, mutexes and file descriptors, nor does it perform any process-level cleanup actions, including, but not limited to, calling any *atexit*( ) routines that may exist.

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 cancellation 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 cancellation 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 1003.1-200x requires that cancellation cleanup handlers be popped and called in order. After all cancellation 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 cancellation

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 1003.1-200x, **<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.

Issue 7

The *pthread_exit*( ) function is moved from the Threads option to the Base.

NAME

pthread_getconcurrency, pthread_setconcurrency — get and set the level of concurrency

SYNOPSIS

OB 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 specified by *new_level* would cause a system resource to be exceeded.

These functions shall not return an error code of [EINTR].

35076 **EXAMPLES**

35077 None.

35078 **APPLICATION USAGE**

35079 Application developers should note that an implementation can always ignore any calls to
35080 *pthread_setconcurrency*( ) and return a constant for *pthread_getconcurrency*( ). For this reason, it is
35081 not recommended that portable applications use this function.

35082 **RATIONALE**

35083 None.

35084 **FUTURE DIRECTIONS**

35085 These functions may be removed in a future version.

35086 **SEE ALSO**

35087 The Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

35088 **CHANGE HISTORY**

35089 First released in Issue 5.

35090 **Issue 7**

35091 The *pthread_getconcurrency*( ) and *pthread_setconcurrency*( ) functions are marked obsolescent.

pthread_getcpuclockid( )

NAME

pthread_getcpuclockid — access a thread CPU-time clock (**ADVANCED REALTIME THREADS**)

SYNOPSIS

TCT

```
#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 1003.1-200x, **<pthread.h>**, **<time.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

Issue 7

The *pthread_getcpuclockid*( ) function is moved from the Threads option.

NAME

pthread_getschedparam, pthread_setschedparam — dynamic thread scheduling parameters access (**REALTIME THREADS**)

SYNOPSIS

TPS

```
#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*.

TSP

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.

35168 **ERRORS**

35169 The *pthread_getschedparam*( ) function may fail if:

35170 [ESRCH] The value specified by *thread* does not refer to an existing thread.

35171 The *pthread_setschedparam*( ) function may fail if:

35172 [EINVAL] The value specified by *policy* or one of the scheduling parameters associated
35173 with the scheduling policy *policy* is invalid.

35174 [ENOTSUP] An attempt was made to set the policy or scheduling parameters to an
35175 unsupported value.

35176 TSP [ENOTSUP] An attempt was made to dynamically change the scheduling policy to
35177 SCHED_SPORADIC, and the implementation does not support this change.

35178 [EPERM] The caller does not have the appropriate permission to set either the
35179 scheduling parameters or the scheduling policy of the specified thread.

35180 [EPERM] The implementation does not allow the application to modify one of the
35181 parameters to the value specified.

35182 [ESRCH] The value specified by *thread* does not refer to a existing thread.

35183 These functions shall not return an error code of [EINTR].

35184 **EXAMPLES**

35185 None.

35186 **APPLICATION USAGE**

35187 None.

35188 **RATIONALE**

35189 None.

35190 **FUTURE DIRECTIONS**

35191 None.

35192 **SEE ALSO**

35193 *pthread_setschedprio*( ), *sched_getparam*( ), *sched_getscheduler*( ), the Base Definitions volume of
35194 IEEE Std 1003.1-200x, **<pthread.h>**, **<sched.h>**

35195 **CHANGE HISTORY**

35196 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

35197 **Issue 6**

35198 The *pthread_getschedparam*( ) and *pthread_setschedparam*( ) functions are marked as part of the
35199 Threads and Thread Execution Scheduling options.

35200 The [ENOSYS] error condition has been removed as stubs need not be provided if an
35201 implementation does not support the Thread Execution Scheduling option.

35202 The Open Group Corrigendum U026/2 is applied, correcting the prototype for the
35203 *pthread_setschedparam*( ) function so that its second argument is of type **int**.

35204 The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

35205 The **restrict** keyword is added to the *pthread_getschedparam*( ) prototype for alignment with the
35206 ISO/IEC 9899: 1999 standard.

35207 The Open Group Corrigendum U047/1 is applied.

35208 IEEE PASC Interpretation 1003.1 #96 is applied, noting that priority values can also be set by a
35209 call to the *pthread_setschedprio*( ) function.

Issue 7

The *pthread_getschedparam*( ) and *pthread_setschedparam*( ) functions are moved from the Threads option.

35213 **NAME**

35214 pthread_getspecific, pthread_setspecific — thread-specific data management

35215 **SYNOPSIS**

35216
```
#include <pthread.h>
```

35217
```
void *pthread_getspecific(pthread_key_t key);
```
35218
```
int pthread_setspecific(pthread_key_t key, const void *value);
```

35219 **DESCRIPTION**

35220 The *pthread_getspecific*( ) function shall return the value currently bound to the specified *key* on
35221 behalf of the calling thread.

35222 The *pthread_setspecific*( ) function shall associate a thread-specific *value* with a *key* obtained via a
35223 previous call to *pthread_key_create*( ). Different threads may bind different values to the same
35224 key. These values are typically pointers to blocks of dynamically allocated memory that have
35225 been reserved for use by the calling thread.

35226 The effect of calling *pthread_getspecific*( ) or *pthread_setspecific*( ) with a *key* value not obtained
35227 from *pthread_key_create*( ) or after *key* has been deleted with *pthread_key_delete*( ) is undefined.

35228 Both *pthread_getspecific*( ) and *pthread_setspecific*( ) may be called from a thread-specific data
35229 destructor function. A call to *pthread_getspecific*( ) for the thread-specific data key being
35230 destroyed shall return the value NULL, unless the value is changed (after the destructor starts)
35231 by a call to *pthread_setspecific*( ). Calling *pthread_setspecific*( ) from a thread-specific data
35232 destructor routine may result either in lost storage (after at least
35233 PTHREAD_DESTRUCTOR_ITERATIONS attempts at destruction) or in an infinite loop.

35234 Both functions may be implemented as macros.

35235 **RETURN VALUE**

35236 The *pthread_getspecific*( ) function shall return the thread-specific data value associated with the
35237 given *key*. If no thread-specific data value is associated with *key*, then the value NULL shall be
35238 returned.

35239 If successful, the *pthread_setspecific*( ) function shall return zero; otherwise, an error number shall
35240 be returned to indicate the error.

35241 **ERRORS**

35242 No errors are returned from *pthread_getspecific*( ).

35243 The *pthread_setspecific*( ) function shall fail if:

35244 [ENOMEM] Insufficient memory exists to associate the non-NULL value with the key.

35245 The *pthread_setspecific*( ) function may fail if:

35246 [EINVAL] The key value is invalid.

35247 The *pthread_setspecific*( ) function shall not return an error code of [EINTR].

35248 **EXAMPLES**

35249 None.

35250 **APPLICATION USAGE**

35251 None.

35252 **RATIONALE**

35253 Performance and ease-of-use of *pthread_getspecific*( ) are critical for functions that rely on
35254 maintaining state in thread-specific data. Since no errors are required to be detected by it, and
35255 since the only error that could be detected is the use of an invalid key, the function to
35256 *pthread_getspecific*( ) has been designed to favor speed and simplicity over error reporting.

35257 **FUTURE DIRECTIONS**

35258 None.

35259 **SEE ALSO**

35260 *pthread_key_create*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

35261 **CHANGE HISTORY**

35262 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

35263 **Issue 6**

35264 The *pthread_getspecific*( ) and *pthread_setspecific*( ) functions are marked as part of the Threads
35265 option.

35266 IEEE PASC Interpretation 1003.1c #3 (Part 6) is applied, updating the DESCRIPTION.

35267 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/96 is applied, updating the ERRORS
35268 section so that the [ENOMEM] error case is changed from "to associate the value with the key"
35269 to "to associate the non-NULL value with the key".

35270 **Issue 7**

35271 Austin Group Interpretation 1003.1-2001 #063 is applied, updating the ERRORS section.

35272 The *pthread_getspecific*( ) and *pthread_setspecific*( ) functions are moved from the Threads option to
35273 the Base.

NAME

pthread_join — wait for thread termination

SYNOPSIS

```
#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:

[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.

[EINVAL] The value specified by *thread* does not refer to a joinable thread.

The *pthread_join*() function shall not return an error code of [EINTR].

EXAMPLES

An example of thread creation and deletion follows:

```
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];
    pthread_t  th1, th2;
    subarray   sb1, sb2;
```

```
        sb1.ar = &ar[0];
        sb1.n  = 500000;
        (void) pthread_create(&th1, NULL, incer, &sb1);

        sb2.ar = &ar[500000];
        sb2.n  = 500000;
        (void) pthread_create(&th2, NULL, incer, &sb2);

        (void) pthread_join(th1, NULL);
        (void) pthread_join(th2, NULL);
        return 0;
    }
```

APPLICATION USAGE

None.

RATIONALE

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 1003.1-200x 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 cancellation 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.
- Cancellation 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 cancellation 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 1003.1-200x, Section 4.10, Memory Synchronization, **<pthread.h>**

CHANGE HISTORY

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

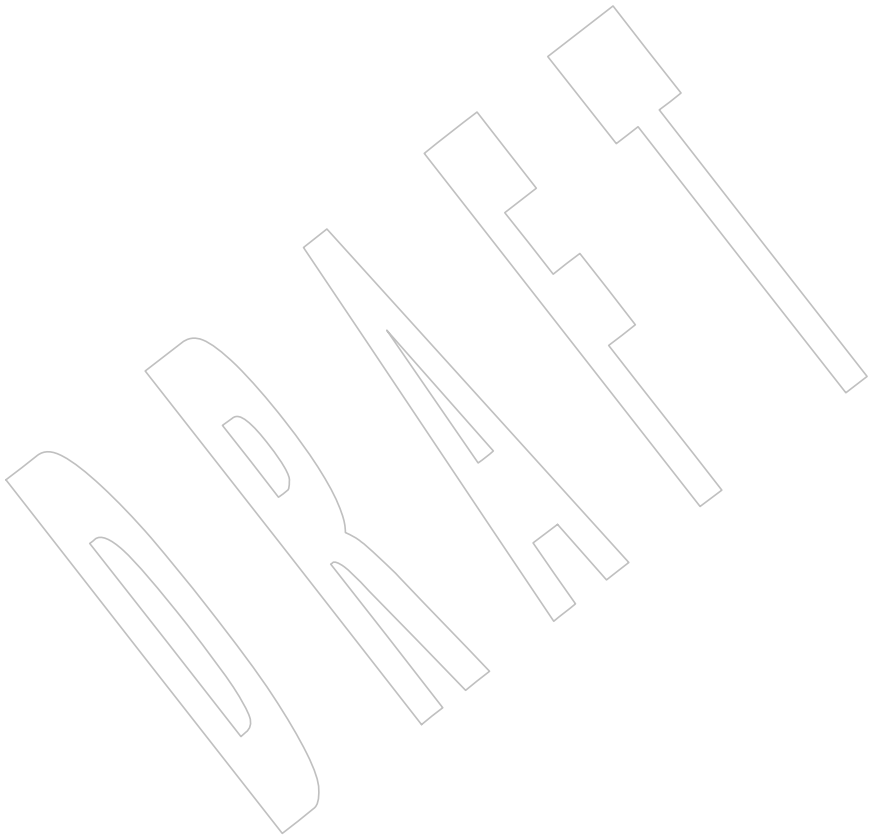
Issue 6

The *pthread_join*( ) function is marked as part of the Threads option.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/97 is applied, updating the ERRORS section so that the [EINVAL] error is made optional and the words ''the implementation has detected'' are removed from it.

Issue 7

The *pthread_join*( ) function is moved from the Threads option to the Base.



35370 **NAME**

35371 pthread_key_create — thread-specific data key creation

35372 **SYNOPSIS**

35373 ```
#include <pthread.h>
```

35374 ```
int pthread_key_create(pthread_key_t *key, void (*destructor)(void*));
```

35375 **DESCRIPTION**

35376 The *pthread_key_create*( ) function shall create a thread-specific data key visible to all threads in
35377 the process. Key values provided by *pthread_key_create*( ) are opaque objects used to locate
35378 thread-specific data. Although the same key value may be used by different threads, the values
35379 bound to the key by *pthread_setspecific*( ) are maintained on a per-thread basis and persist for the
35380 life of the calling thread.

35381 Upon key creation, the value NULL shall be associated with the new key in all active threads.
35382 Upon thread creation, the value NULL shall be associated with all defined keys in the new
35383 thread.

35384 An optional destructor function may be associated with each key value. At thread exit, if a key
35385 value has a non-NULL destructor pointer, and the thread has a non-NULL value associated with
35386 that key, the value of the key is set to NULL, and then the function pointed to is called with the
35387 previously associated value as its sole argument. The order of destructor calls is unspecified if
35388 more than one destructor exists for a thread when it exits.

35389 If, after all the destructors have been called for all non-NULL values with associated destructors,
35390 there are still some non-NULL values with associated destructors, then the process is repeated.
35391 If, after at least {PTHREAD_DESTRUCTOR_ITERATIONS} iterations of destructor calls for
35392 outstanding non-NULL values, there are still some non-NULL values with associated
35393 destructors, implementations may stop calling destructors, or they may continue calling
35394 destructors until no non-NULL values with associated destructors exist, even though this might
35395 result in an infinite loop.

35396 **RETURN VALUE**

35397 If successful, the *pthread_key_create*( ) function shall store the newly created key value at **key* and
35398 shall return zero. Otherwise, an error number shall be returned to indicate the error.

35399 **ERRORS**

35400 The *pthread_key_create*( ) function shall fail if:

35401 [EAGAIN] The system lacked the necessary resources to create another thread-specific
35402 data key, or the system-imposed limit on the total number of keys per process
35403 {PTHREAD_KEYS_MAX} has been exceeded.

35404 [ENOMEM] Insufficient memory exists to create the key.

35405 The *pthread_key_create*( ) function shall not return an error code of [EINTR].

35406 **EXAMPLES**

35407 The following example demonstrates a function that initializes a thread-specific data key when it
35408 is first called, and associates a thread-specific object with each calling thread, initializing this
35409 object when necessary.

35410 ```
static pthread_key_t key;
```
35411 ```
static pthread_once_t key_once = PTHREAD_ONCE_INIT;
```

35412 ```
static void
```
35413 ```
make_key()
```
35414 ```
{
```

  
```

```
        (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.

```
    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:

```
pthread_getspecific(key, &old);
new = allocate();
destructor(old);
pthread_setspecific(key, new);
```

Note: The above example could leak storage if run with asynchronous cancellation enabled. No such problems occur in the default cancellation state if no cancellation 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 1003.1-200x, **<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.

Issue 7

The *pthread_key_create*( ) function is moved from the Threads option to the Base.

NAME

pthread_key_delete — thread-specific data key deletion

SYNOPSIS

```
#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 1003.1-200x, **<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.

Issue 7

The *pthread_key_delete( )* function is moved from the Threads option to the Base.

35561 **NAME**

35562 pthread_kill — send a signal to a thread

35563 **SYNOPSIS**

35564 CX
```
#include <signal.h>

int pthread_kill(pthread_t thread, int sig);
```

35566 **DESCRIPTION**

35567 The *pthread_kill*( ) function shall request that a signal be delivered to the specified thread.

35568 As in *kill*( ), if *sig* is zero, error checking shall be performed but no signal shall actually be sent.

35569 **RETURN VALUE**

35570 Upon successful completion, the function shall return a value of zero. Otherwise, the function
35571 shall return an error number. If the *pthread_kill*( ) function fails, no signal shall be sent.

35572 **ERRORS**

35573 The *pthread_kill*( ) function shall fail if:

35574 [ESRCH] No thread could be found corresponding to that specified by the given thread
35575 ID.

35576 [EINVAL] The value of the *sig* argument is an invalid or unsupported signal number.

35577 The *pthread_kill*( ) function shall not return an error code of [EINTR].

35578 **EXAMPLES**

35579 None.

35580 **APPLICATION USAGE**

35581 The *pthread_kill*( ) function provides a mechanism for asynchronously directing a signal at a
35582 thread in the calling process. This could be used, for example, by one thread to affect broadcast
35583 delivery of a signal to a set of threads.

35584 Note that *pthread_kill*( ) only causes the signal to be handled in the context of the given thread;
35585 the signal action (termination or stopping) affects the process as a whole.

35586 **RATIONALE**

35587 None.

35588 **FUTURE DIRECTIONS**

35589 None.

35590 **SEE ALSO**

35591 *kill*( ), *pthread_self*( ), *raise*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

35592 **CHANGE HISTORY**

35593 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

35594 **Issue 6**

35595 The *pthread_kill*( ) function is marked as part of the Threads option.

35596 The APPLICATION USAGE section is added.

35597 **Issue 7**

35598 The *pthread_kill*( ) function is moved from the Threads option to the Base.

NAME

pthread_mutex_consistent — mark state protected by robust mutex as consistent

SYNOPSIS

```
#include <pthread.h>

int pthread_mutex_consistent(pthread_mutex_t *mutex);
```

DESCRIPTION

If *mutex* is a robust mutex in an inconsistent state, the *pthread_mutex_consistent*( ) function can be used to mark the state protected by the mutex referenced by *mutex* as consistent again.

If an owner of a robust mutex terminates while holding the mutex, the mutex becomes inconsistent and the next thread that acquires the mutex lock shall be notified of the state by the return value [EOWNERDEAD]. In this case, the mutex does not become normally usable again until the state is marked consistent.

If the thread which acquired the mutex lock with the return value [EOWNERDEAD] terminates before calling either *pthread_mutex_consistent*( ) or *pthread_mutex_unlock*( ), the next thread that acquires the mutex lock shall be notified about the state of the mutex by the return value [EOWNERDEAD].

RETURN VALUE

Upon successful completion, the *pthread_mutex_consistent*( ) function shall return zero. Otherwise, an error value shall be returned to indicate the error.

ERRORS

The *pthread_mutex_consistent*( ) function shall fail if:

[EINVAL] The mutex object referenced by *mutex* is not robust or does not protect an inconsistent state.

The *pthread_mutex_consistent*( ) function may fail if:

[EINVAL] The value *mutex* is invalid.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

The *pthread_mutex_consistent*( ) function is only responsible for notifying the implementation that the state protected by the mutex has been recovered and that normal operations with the mutex can be resumed. It is the responsibility of the application to recover the state so it can be reused. If the application is not able to perform the recovery, it can notify the implementation that the situation is unrecoverable by a call to *pthread_mutex_unlock*( ) without a prior call to *pthread_mutex_consistent*( ), in which case subsequent threads that attempt to lock the mutex will fail to acquire the lock and be returned [ENOTRECOVERABLE].

RATIONALE

None.

FUTURE DIRECTIONS

None.

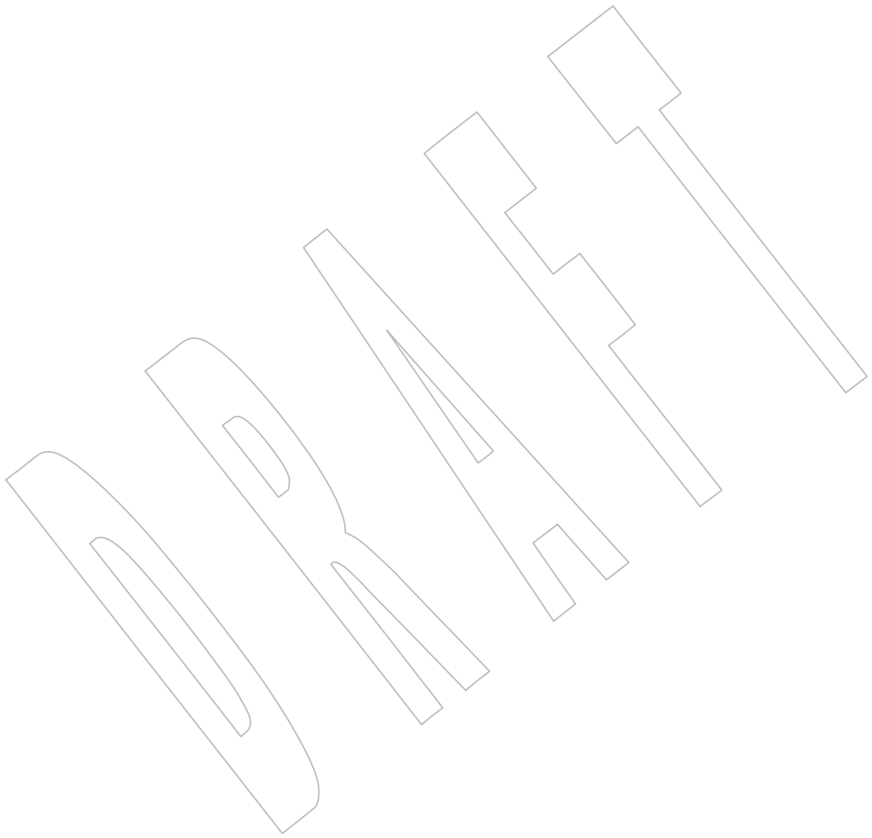
SEE ALSO

pthread_mutex_lock( ), *pthread_mutexattr_getrobust( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 7.

The *pthread_mutex_consistent( )* function is moved from the Threads option to the Base.



NAME

pthread_mutex_destroy, pthread_mutex_init — destroy and initialize a mutex

SYNOPSIS

```
#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.

[ENOMEM] Insufficient memory exists to initialize the mutex.

[EPERM] The caller does not have the privilege to perform the operation.

The *pthread_mutex_init*( ) function may fail if:

[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.

[EINVAL] The attributes object referenced by *attr* has the robust mutex attribute set without the process-shared attribute being set.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Alternate Implementations Possible

This volume of IEEE Std 1003.1-200x 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 1003.1-200x 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 1003.1-200x ensures that a fully-conforming application is portable

across the full range of implementations, while not forcing all implementations to add overhead to check for numerous things that a correct program never does.

Why No Limits are Defined

Defining symbols for the maximum number of mutexes and condition variables was considered but rejected because the number of these objects may change dynamically. Furthermore, many implementations place these objects into application memory; thus, there is no explicit maximum.

Static Initializers for Mutexes and Condition Variables

Providing for static initialization of statically allocated synchronization objects allows modules with private static synchronization variables to avoid runtime initialization tests and overhead. Furthermore, it simplifies the coding of self-initializing modules. Such modules are common in C libraries, where for various reasons the design calls for self-initialization instead of requiring an explicit module initialization function to be called. An example use of static initialization follows.

Without static initialization, a self-initializing routine *foo*( ) might look as follows:

```
static pthread_once_t foo_once = PTHREAD_ONCE_INIT;
static pthread_mutex_t foo_mutex;

void foo_init()
{
    pthread_mutex_init(&foo_mutex, NULL);
}

void foo()
{
    pthread_once(&foo_once, foo_init);
    pthread_mutex_lock(&foo_mutex);
   /* Do work. */
    pthread_mutex_unlock(&foo_mutex);
}
```

With static initialization, the same routine could be coded as follows:

```
static pthread_mutex_t foo_mutex = PTHREAD_MUTEX_INITIALIZER;

void foo()
{
    pthread_mutex_lock(&foo_mutex);
   /* Do work. */
    pthread_mutex_unlock(&foo_mutex);
}
```

Note that the static initialization both eliminates the need for the initialization test inside *pthread_once*( ) and the fetch of *&foo_mutex* to learn the address to be passed to *pthread_mutex_lock*( ) or *pthread_mutex_unlock*( ).

Thus, the C code written to initialize static objects is simpler on all systems and is also faster on a large class of systems; those where the (entire) synchronization object can be stored in application memory.

Yet the locking performance question is likely to be raised for machines that require mutexes to be allocated out of special memory. Such machines actually have to have mutexes and possibly condition variables contain pointers to the actual hardware locks. For static initialization to work on such machines, *pthread_mutex_lock*( ) also has to test whether or not the pointer to the actual

lock has been allocated. If it has not, *pthread_mutex_lock*( ) has to initialize it before use. The reservation of such resources can be made when the program is loaded, and hence return codes have not been added to mutex locking and condition variable waiting to indicate failure to complete initialization.

This runtime test in *pthread_mutex_lock*( ) would at first seem to be extra work; an extra test is required to see whether the pointer has been initialized. On most machines this would actually be implemented as a fetch of the pointer, testing the pointer against zero, and then using the pointer if it has already been initialized. While the test might seem to add extra work, the extra effort of testing a register is usually negligible since no extra memory references are actually done. As more and more machines provide caches, the real expenses are memory references, not instructions executed.

Alternatively, depending on the machine architecture, there are often ways to eliminate *all* overhead in the most important case: on the lock operations that occur *after* the lock has been initialized. This can be done by shifting more overhead to the less frequent operation: initialization. Since out-of-line mutex allocation also means that an address has to be dereferenced to find the actual lock, one technique that is widely applicable is to have static initialization store a bogus value for that address; in particular, an address that causes a machine fault to occur. When such a fault occurs upon the first attempt to lock such a mutex, validity checks can be done, and then the correct address for the actual lock can be filled in. Subsequent lock operations incur no extra overhead since they do not "fault". This is merely one technique that can be used to support static initialization, while not adversely affecting the performance of lock acquisition. No doubt there are other techniques that are highly machine-dependent.

The locking overhead for machines doing out-of-line mutex allocation is thus similar for modules being implicitly initialized, where it is improved for those doing mutex allocation entirely inline. The inline case is thus made much faster, and the out-of-line case is not significantly worse.

Besides the issue of locking performance for such machines, a concern is raised that it is possible that threads would serialize contending for initialization locks when attempting to finish initializing statically allocated mutexes. (Such finishing would typically involve taking an internal lock, allocating a structure, storing a pointer to the structure in the mutex, and releasing the internal lock.) First, many implementations would reduce such serialization by hashing on the mutex address. Second, such serialization can only occur a bounded number of times. In particular, it can happen at most as many times as there are statically allocated synchronization objects. Dynamically allocated objects would still be initialized via *pthread_mutex_init*( ) or *pthread_cond_init*( ).

Finally, if none of the above optimization techniques for out-of-line allocation yields sufficient performance for an application on some implementation, the application can avoid static initialization altogether by explicitly initializing all synchronization objects with the corresponding *pthread_*_init*( ) functions, which are supported by all implementations. An implementation can also document the tradeoffs and advise which initialization technique is more efficient for that particular implementation.

Destroying Mutexes

A mutex can be destroyed immediately after it is unlocked. For example, consider the following code:

```
struct obj {
pthread_mutex_t om;
    int refcnt;
    ...
};
```

```
obj_done(struct obj *op)
{
    pthread_mutex_lock(&op->om);
    if (--op->refcnt == 0) {
        pthread_mutex_unlock(&op->om);
(A)     pthread_mutex_destroy(&op->om);
(B)     free(op);
    } else
(C)     pthread_mutex_unlock(&op->om);
}
```

In this case *obj* is reference counted and *obj_done*( ) is called whenever a reference to the object is dropped. Implementations are required to allow an object to be destroyed and freed and potentially unmapped (for example, lines A and B) immediately after the object is unlocked (line C).

Robust Mutexes

Implementations are required to provide robust mutexes for mutexes with the process-shared attribute set to PTHREAD_PROCESS_SHARED. Implementations are allowed, but not required, to provide robust mutexes when the process-shared attribute is set to PTHREAD_PROCESS_PRIVATE.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_mutex_getprioceiling( ), *pthread_mutexattr_getrobust*( ), *pthread_mutex_lock*( ), *pthread_mutex_timedlock*( ), *pthread_mutexattr_getpshared*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

Issue 7

Changes are made from The Open Group Technical Standard, 2006, Extended API Set Part 3.

The *pthread_mutex_destroy*( ) and *pthread_mutex_init*( ) functions are moved from the Threads option to the Base.

NAME

pthread_mutex_getprioceiling, pthread_mutex_setprioceiling — get and set the priority ceiling of a mutex (**REALTIME THREADS**)

SYNOPSIS

RPP | TPP

```
#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 *pthread_mutex_setprioceiling*( ) is called while holding the mutex, the result is undefined unless the mutex is of type PTHREAD_MUTEX_RECURSIVE.

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

These functions shall fail if:

[EINVAL] The protocol attribute of *mutex* is PTHREAD_PRIO_NONE.

These functions may fail if:

[EDEADLK] The current thread already owns the mutex.

[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 1003.1-200x, **<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_mutex_getprioceiling*( ) and *pthread_mutex_setprioceiling*( ) functions are marked as part of the Threads and Thread Priority Protection options.

The [ENOSYS] error conditions have been removed.

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.

Issue 7

SD5-XSH-ERN-39 is applied.

Austin Group Interpretation 1003.1-2001 #052 is applied, adding [EDEADLK] as a "may fail" error.

SD5-XSH-ERN-158 is applied, updating the ERRORS section to include a "shall fail" error case for when the protocol attribute of *mutex* is PTHREAD_PRIO_NONE.

The *pthread_mutex_getprioceiling*( ) and *pthread_mutex_setprioceiling*( ) functions are moved from the Threads option to require support of either the Robust Mutex Priority Protection option or the Non-Robust Mutex Priority Protection option.

35930 **NAME**

35931 pthread_mutex_init — destroy and initialize a mutex

35932 **SYNOPSIS**

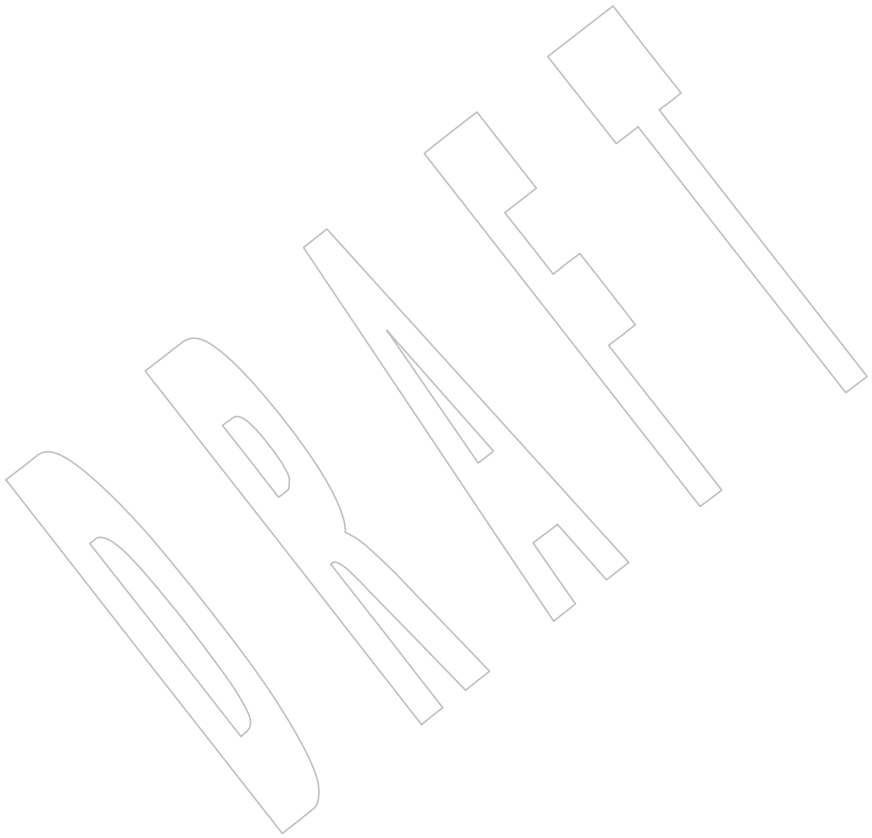
35933
```
#include <pthread.h>
```

35934
```
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
    const pthread_mutexattr_t *restrict attr);
```

35936
```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

35937 **DESCRIPTION**

35938 Refer to *pthread_mutex_destroy( )*.



35939 **NAME**

35940 pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock — lock and unlock a
35941 mutex

35942 **SYNOPSIS**

35943 ```
#include <pthread.h>
```

35944 ```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```
35945 ```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```
35946 ```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

35947 **DESCRIPTION**

35948 The mutex object referenced by *mutex* shall be locked by calling *pthread_mutex_lock*( ). If the
35949 mutex is already locked, the calling thread shall block until the mutex becomes available. This
35950 operation shall return with the mutex object referenced by *mutex* in the locked state with the
35951 calling thread as its owner.

35952 If the mutex type is PTHREAD_MUTEX_NORMAL, deadlock detection shall not be provided.
35953 Attempting to relock the mutex causes deadlock. If a thread attempts to unlock a mutex that it
35954 has not locked or a mutex which is unlocked, undefined behavior results.

35955 If the mutex type is PTHREAD_MUTEX_ERRORCHECK, then error checking shall be provided.
35956 If a thread attempts to relock a mutex that it has already locked, an error shall be returned. If a
35957 thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error
35958 shall be returned.

35959 If the mutex type is PTHREAD_MUTEX_RECURSIVE, then the mutex shall maintain the
35960 concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock
35961 count shall be set to one. Every time a thread relocks this mutex, the lock count shall be
35962 incremented by one. Each time the thread unlocks the mutex, the lock count shall be
35963 decremented by one. When the lock count reaches zero, the mutex shall become available for
35964 other threads to acquire. If a thread attempts to unlock a mutex that it has not locked or a mutex
35965 which is unlocked, an error shall be returned.

35966 If the mutex type is PTHREAD_MUTEX_DEFAULT, attempting to recursively lock the mutex
35967 results in undefined behavior. Attempting to unlock the mutex if it was not locked by the calling
35968 thread results in undefined behavior. Attempting to unlock the mutex if it is not locked results in
35969 undefined behavior.

35970 The *pthread_mutex_trylock*( ) function shall be equivalent to *pthread_mutex_lock*( ), except that if
35971 the mutex object referenced by *mutex* is currently locked (by any thread, including the current
35972 thread), the call shall return immediately. If the mutex type is PTHREAD_MUTEX_RECURSIVE
35973 and the mutex is currently owned by the calling thread, the mutex lock count shall be
35974 incremented by one and the *pthread_mutex_trylock*( ) function shall immediately return success.

35975 The *pthread_mutex_unlock*( ) function shall release the mutex object referenced by *mutex*. The
35976 manner in which a mutex is released is dependent upon the mutex's type attribute. If there are
35977 threads blocked on the mutex object referenced by *mutex* when *pthread_mutex_unlock*( ) is called,
35978 resulting in the mutex becoming available, the scheduling policy shall determine which thread
35979 shall acquire the mutex.

35980 (In the case of PTHREAD_MUTEX_RECURSIVE mutexes, the mutex shall become available
35981 when the count reaches zero and the calling thread no longer has any locks on this mutex.)

35982 If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the
35983 thread shall resume waiting for the mutex as if it was not interrupted.

35984 If *mutex* is a robust mutex and the process containing the owning thread terminated while

holding the mutex lock, a call to *pthread_mutex_lock*( ) shall return the error value [EOWNERDEAD]. If *mutex* is a robust mutex and the owning thread terminated while holding the mutex lock, a call to *pthread_mutex_lock*( ) may return the error value [EOWNERDEAD] even if the process in which the owning thread resides has not terminated. In these cases, the mutex is locked by the thread but the state it protects is marked as inconsistent. The application should ensure that the state is made consistent for reuse and when that is complete call *pthread_mutex_consistent*( ). If the application is unable to recover the state, it should unlock the mutex without a prior call to *pthread_mutex_consistent*( ), after which the mutex is marked permanently unusable.

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:

RPP | TPP [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.

[ENOTRECOVERABLE]
The state protected by the mutex is not recoverable. The mutex is not locked.

[EOWNERDEAD]
The mutex is a robust mutex and the process containing the previous owning thread terminated while holding the mutex lock. The mutex lock has been acquired and it is up to the new owner to make the state consistent.

The *pthread_mutex_trylock*( ) function shall fail if:

[EBUSY] The *mutex* could not be acquired because it was already locked.

The *pthread_mutex_unlock*( ) function shall fail if:

[EPERM] The current thread does not own the mutex and the mutex is a robust mutex.

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.

The *pthread_mutex_lock*( ) and *pthread_mutex_trylock*( ) functions may fail if:

[EOWNERDEAD]
The mutex is a robust mutex and the previous owning thread terminated while holding the mutex lock. The mutex lock has been acquired and it is up to the new owner to make the state consistent.

The *pthread_mutex_lock*( ) function may fail if:

[EDEADLK] A deadlock condition was detected or the current thread already owns the mutex.

The *pthread_mutex_unlock*( ) function may fail if:

[EPERM] The current thread does not own the mutex.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

Applications that have assumed that non-zero return values are errors will need updating for use with robust mutexes, since a valid return for a thread acquiring a mutex which is protecting a currently inconsistent state is [EOWNERDEAD]. Applications that do not check the error returns, due to ruling out the possibility of such errors arising, should not use robust mutexes. If an application is supposed to work with normal and robust mutexes it should check all return values for error conditions and if necessary take appropriate action.

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.

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.

For further rationale on the extended mutex types, see the Rationale (Informative) volume of IEEE Std 1003.1-200x.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_mutex_consistent( ), *pthread_mutex_destroy( )*, *pthread_mutex_timedlock( )*, *pthread_mutexattr_getrobust( )*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.10, Memory Synchronization, **<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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/98 is applied, updating the ERRORS section so that the [EDEADLK] error includes detection of a deadlock condition. The RATIONALE section is also reworded to take into account non-XSI-conformant systems.

Issue 7

SD5-XSH-ERN-43 is applied, marking the "shall fail" case of the [EINVAL] error as dependent on the Thread Priority Protection option.

Changes are made from The Open Group Technical Standard, 2006, Extended API Set Part 3.

The *pthread_mutex_lock*(), *pthread_mutex_trylock*(), and *pthread_mutex_unlock*() functions are moved from the Threads option to the Base.

The PTHREAD_MUTEX_NORMAL, PTHREAD_MUTEX_ERRORCHECK, PTHREAD_MUTEX_RECURSIVE, and PTHREAD_MUTEX_DEFAULT extended mutex types are moved from the XSI option to the Base.

36081 **NAME**

36082 pthread_mutex_setprioceiling — change the priority ceiling of a mutex (**REALTIME**
36083 **THREADS**)

36084 **SYNOPSIS**

36085 RPP | TPP
```
#include <pthread.h>

int pthread_mutex_setprioceiling(pthread_mutex_t *restrict mutex,
    int prioceiling, int *restrict old_ceiling);
```

36088 **DESCRIPTION**

36089 Refer to *pthread_mutex_getprioceiling( )*.

NAME

pthread_mutex_timedlock — lock a mutex

SYNOPSIS

```
#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.

The timeout shall be based on the CLOCK_REALTIME clock. 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.

RPI | TPI 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.

If *mutex* is a robust mutex and the process containing the owning thread terminated while holding the mutex lock, a call to *pthread_mutex_timedlock*( ) shall return the error value [EOWNERDEAD]. If *mutex* is a robust mutex and the owning thread terminated while holding the mutex lock, a call to *pthread_mutex_timedlock*( ) may return the error value [EOWNERDEAD] even if the process in which the owning thread resides has not terminated. In these cases, the mutex is locked by the thread but the state it protects is marked as inconsistent. The application should ensure that the state is made consistent for reuse and when that is complete call *pthread_mutex_consistent*( ). If the application is unable to recover the state, it should unlock the mutex without a prior call to *pthread_mutex_consistent*( ), after which the mutex is marked permanently unusable.

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.

[ENOTRECOVERABLE]
The state protected by the mutex is not recoverable. The mutex is not locked.

[EOWNERDEAD]
The mutex is a robust mutex and the process containing the previous owning thread terminated while holding the mutex lock. The mutex lock has been acquired and it is up to the new owner to make the state consistent.

[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.

[EAGAIN] The mutex could not be acquired because the maximum number of recursive locks for *mutex* has been exceeded.

[EDEADLK] A deadlock condition was detected or the current thread already owns the mutex.

[EOWNERDEAD]
The mutex is a robust mutex and the previous owning thread terminated while holding the mutex lock. The mutex lock has been acquired and it is up to the new owner to make the state consistent.

This function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

Applications that have assumed that non-zero return values are errors will need updating for use with robust mutexes, since a valid return for a thread acquiring a mutex which is protecting a currently inconsistent state is [EOWNERDEAD]. Applications that do not check the error returns, due to ruling out the possibility of such errors arising, should not use robust mutexes. If an application is supposed to work with normal and robust mutexes, it should check all return values for error conditions and if necessary take appropriate action.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_mutex_destroy( ), *pthread_mutex_lock*( ), *pthread_mutex_trylock*( ), *time*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.10, Memory Synchronization, **<pthread.h>**, **<time.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/99 is applied, marking the last paragraph in the DESCRIPTION as part of the Thread Priority Inheritance option.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/100 is applied, updating the ERRORS section so that the [EDEADLK] error includes detection of a deadlock condition.

Issue 7

Changes are made from The Open Group Technical Standard, 2006, Extended API Set Part 3.

The *pthread_mutex_timedlock*( ) function is moved from the Timeouts option to the Base.

Functionality relating to the Timers option is moved to the Base.

36182 **NAME**

36183 pthread_mutex_trylock, pthread_mutex_unlock — lock and unlock a mutex

36184 **SYNOPSIS**

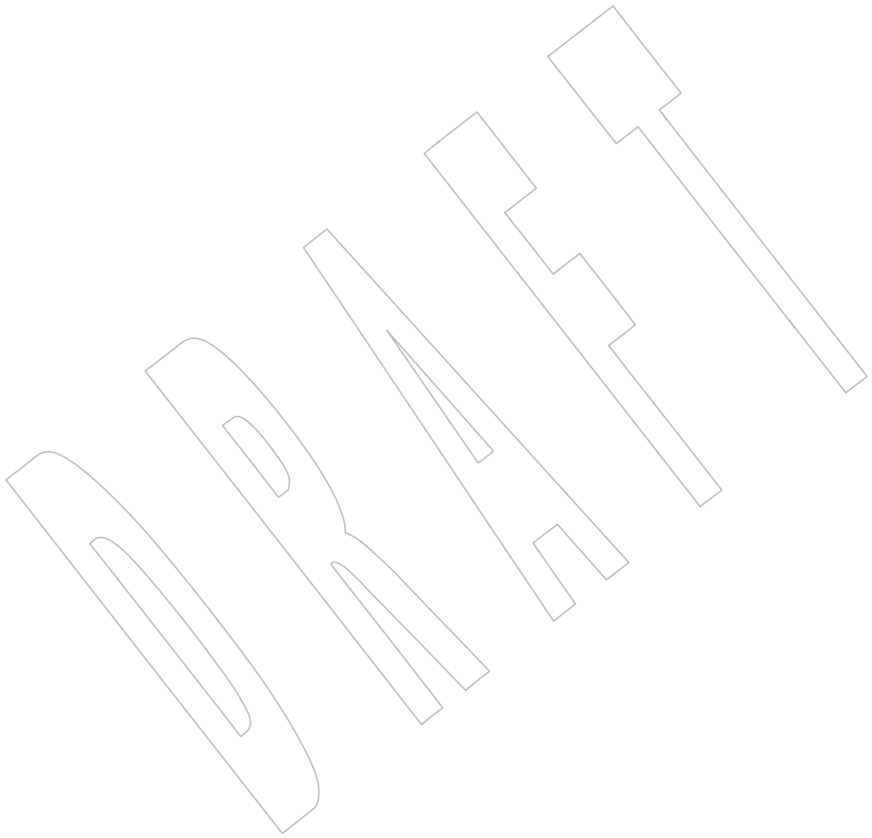
36185
```
#include <pthread.h>
```

36186
```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

36187
```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

36188 **DESCRIPTION**

36189 Refer to *pthread_mutex_lock( )*.



NAME

pthread_mutexattr_destroy, pthread_mutexattr_init — destroy and initialize the mutex attributes object

SYNOPSIS

```
#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 1003.1-200x without changing the existing functions. Thus, they provide for future extensibility of this volume of IEEE Std 1003.1-200x 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 1003.1-200x. 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 1003.1-200x leads to the possibility that an application may allocate the synchronization objects from this section in memory that is accessed by multiple processes (and therefore, by threads of multiple processes).

In order to permit such usage, while at the same time keeping the usual case (that is, usage within a single process) efficient, a *process-shared* option has been defined.

If an implementation supports the _POSIX_THREAD_PROCESS_SHARED option, then the *process-shared* attribute can be used to indicate that mutexes or condition variables may be accessed by threads of multiple processes.

The default setting of PTHREAD_PROCESS_PRIVATE has been chosen for the *process-shared* attribute so that the most efficient forms of these synchronization objects are created by default.

Synchronization variables that are initialized with the PTHREAD_PROCESS_PRIVATE *process-shared* attribute may only be operated on by threads in the process that initialized them. Synchronization variables that are initialized with the PTHREAD_PROCESS_SHARED *process-shared* attribute may be operated on by any thread in any process that has access to it. In particular, these processes may exist beyond the lifetime of the initializing process. For example, the following code implements a simple counting semaphore in a mapped file that may be used by many processes.

```
/* sem.h */
struct semaphore {
    pthread_mutex_t lock;
    pthread_cond_t nonzero;
    unsigned count;
};
typedef struct semaphore semaphore_t;

semaphore_t *semaphore_create(char *semaphore_name);
semaphore_t *semaphore_open(char *semaphore_name);
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);
    return (semap);
}

void
semaphore_post(semaphore_t *semap)
{
    pthread_mutex_lock(&semap->lock);
    if (semap->count == 0)
```

```
        pthread_cond_signal(&semapx->nonzero);
    semap->count++;
    pthread_mutex_unlock(&semap->lock);
}

void
semaphore_wait(semaphore_t *semap)
{
    pthread_mutex_lock(&semap->lock);
    while (semap->count == 0)
        pthread_cond_wait(&semap->nonzero, &semap->lock);
    semap->count--;
    pthread_mutex_unlock(&semap->lock);
}

void
semaphore_close(semaphore_t *semap)
{
    munmap((void *) semap, sizeof(semaphore_t));
}
```

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.

```
/* create.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);
}

/* post */
#include "pthread.h"
#include "sem.h"

int
main()
{
    semaphore_t *semap;

    semap = semaphore_open("/tmp/semaphore");
    if (semap == NULL)
        exit(1);
    semaphore_post(semap);
    semaphore_close(semap);
    return (0);
}
```



```
/* wait */
#include "pthread.h"
#include "sem.h"

int
main()
{
    semaphore_t *semap;

    semap = semaphore_open("/tmp/semaphore");
    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 Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

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.

Issue 7

The *pthread_mutexattr_destroy*( ) and *pthread_mutexattr_init*( ) functions are moved from the Threads option to the Base.

36411 **NAME**

36412 pthread_mutexattr_getprioceiling, pthread_mutexattr_setprioceiling — get and set the
36413 prioceiling attribute of the mutex attributes object (**REALTIME THREADS**)

36414 **SYNOPSIS**

36415 RPP | TPP
```
#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);
```

36420 **DESCRIPTION**

36421 The *pthread_mutexattr_getprioceiling*( ) and *pthread_mutexattr_setprioceiling*( ) functions,
36422 respectively, shall get and set the priority ceiling attribute of a mutex attributes object pointed to
36423 by *attr* which was previously created by the function *pthread_mutexattr_init*( ).

36424 The *prioceiling* attribute contains the priority ceiling of initialized mutexes. The values of
36425 *prioceiling* are within the maximum range of priorities defined by SCHED_FIFO.

36426 The *prioceiling* attribute defines the priority ceiling of initialized mutexes, which is the minimum
36427 priority level at which the critical section guarded by the mutex is executed. In order to avoid
36428 priority inversion, the priority ceiling of the mutex shall be set to a priority higher than or equal
36429 to the highest priority of all the threads that may lock that mutex. The values of *prioceiling* are
36430 within the maximum range of priorities defined under the SCHED_FIFO scheduling policy.

36431 **RETURN VALUE**

36432 Upon successful completion, the *pthread_mutexattr_getprioceiling*( ) and
36433 *pthread_mutexattr_setprioceiling*( ) functions shall return zero; otherwise, an error number shall be
36434 returned to indicate the error.

36435 **ERRORS**

36436 The *pthread_mutexattr_getprioceiling*( ) and *pthread_mutexattr_setprioceiling*( ) functions may fail if:

36437 [EINVAL] The value specified by *attr* or *prioceiling* is invalid.

36438 [EPERM] The caller does not have the privilege to perform the operation.

36439 These functions shall not return an error code of [EINTR].

36440 **EXAMPLES**

36441 None.

36442 **APPLICATION USAGE**

36443 None.

36444 **RATIONALE**

36445 None.

36446 **FUTURE DIRECTIONS**

36447 None.

36448 **SEE ALSO**

36449 *pthread_cond_destroy*( ), *pthread_create*( ), *pthread_mutex_destroy*( ), the Base Definitions volume of
36450 IEEE Std 1003.1-200x, **<pthread.h>**

36451 **CHANGE HISTORY**

36452 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

36453 Marked as part of the Realtime Threads Feature Group.

36454 **Issue 6**

36455 The *pthread_mutexattr_getprioceiling*( ) and *pthread_mutexattr_setprioceiling*( ) functions are marked
36456 as part of the Threads and Thread Priority Protection options.

36457 The [ENOSYS] error condition has been removed as stubs need not be provided if an
36458 implementation does not support the Thread Priority Protection option.

36459 The [ENOTSUP] error condition has been removed since these functions do not have a *protocol*
36460 argument.

36461 The **restrict** keyword is added to the *pthread_mutexattr_getprioceiling*( ) prototype for alignment
36462 with the ISO/IEC 9899: 1999 standard.

36463 **Issue 7**

36464 The *pthread_mutexattr_getprioceiling*( ) and *pthread_mutexattr_setprioceiling*( ) functions are moved
36465 from the Threads option to require support of either the Robust Mutex Priority Protection option
36466 or the Non-Robust Mutex Priority Protection option.

NAME

pthread_mutexattr_getprotocol, pthread_mutexattr_setprotocol — get and set the protocol attribute of the mutex attributes object (**REALTIME THREADS**)

SYNOPSIS

MC1
```
#include <pthread.h>

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:

RPI | TPI PTHREAD_PRIO_INHERIT
MC1 PTHREAD_PRIO_NONE
RPP | TPP PTHREAD_PRIO_PROTECT

which are defined in the **<pthread.h>** header. The default value of the attribute shall be PTHREAD_PRIO_NONE.

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.

RPI When a thread is blocking higher priority threads because of owning one or more robust 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 robust mutexes owned by this thread and initialized with this protocol.

TPI When a thread is blocking higher priority threads because of owning one or more non-robust 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 non-robust mutexes owned by this thread and initialized with this protocol.

RPP When a thread owns one or more robust 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 robust mutexes owned by this thread and initialized with this attribute, regardless of whether other threads are blocked on any of these robust mutexes or not.

TPP When a thread owns one or more non-robust 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 non-robust mutexes owned by this thread and initialized with this attribute, regardless of whether other threads are blocked on any of these non-robust 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.

RPI | TPI 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 inheritance 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 [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 1003.1-200x, **<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.

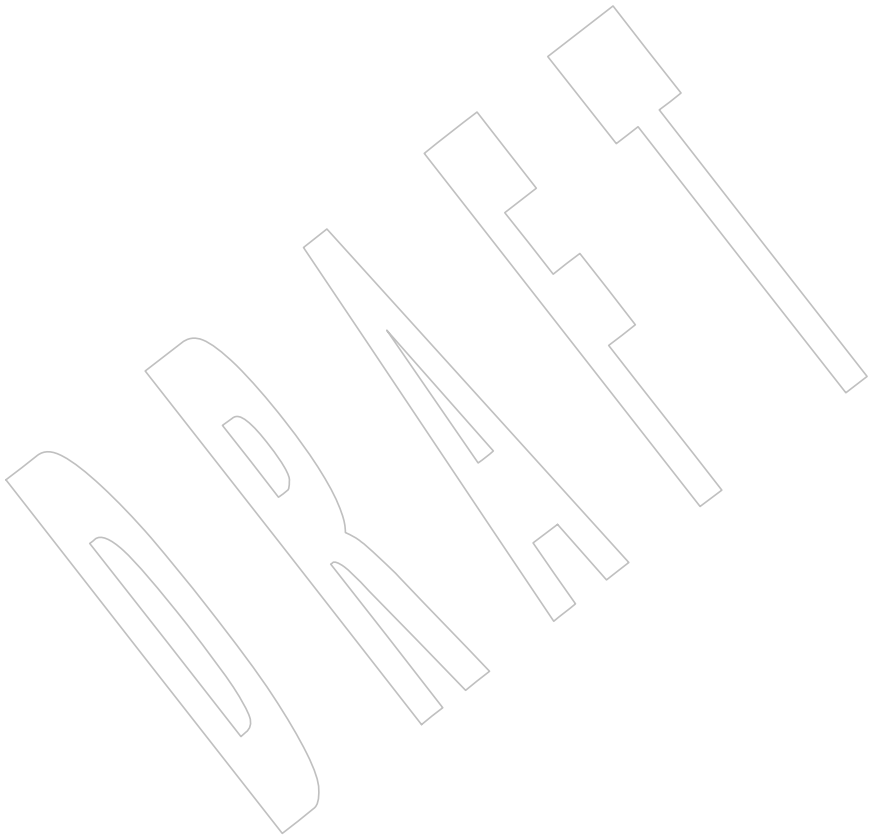
The **restrict** keyword is added to the *pthread_mutexattr_getprotocol*( ) prototype for alignment

with the ISO/IEC 9899: 1999 standard.

Issue 7

The *pthread_mutexattr_getprotocol*( ) and *pthread_mutexattr_setprotocol*( ) functions are moved from the Threads option to require support of either the Non-Robust Mutex Priority Protection option or the Non-Robust Mutex Priority Inheritance option or the Robust Mutex Priority Protection option or the Robust Mutex Priority Inheritance option.

SD5-XSH-ERN-135 is applied, updating the DESCRIPTION to define a default value for the *protocol* attribute.



NAME

pthread_mutexattr_getpshared, pthread_mutexattr_setpshared — get and set the process-shared attribute

SYNOPSIS

TSH

```
#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 [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

36602 **FUTURE DIRECTIONS**

36603 None.

36604 **SEE ALSO**

36605 *pthread_cond_destroy( )*, *pthread_create( )*, *pthread_mutex_destroy( )*, *pthread_mutexattr_destroy( )*, the
36606 Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

36607 **CHANGE HISTORY**

36608 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

36609 **Issue 6**

36610 The *pthread_mutexattr_getpshared*( ) and *pthread_mutexattr_setpshared*( ) functions are marked as
36611 part of the Threads and Thread Process-Shared Synchronization options.

36612 The **restrict** keyword is added to the *pthread_mutexattr_getpshared*( ) prototype for alignment
36613 with the ISO/IEC 9899: 1999 standard.

36614 **Issue 7**

36615 The *pthread_mutexattr_getpshared*( ) and *pthread_mutexattr_setpshared*( ) functions are moved from
36616 the Threads option.

NAME

pthread_mutexattr_getrobust, pthread_mutexattr_setrobust — get and set the mutex robust attribute

SYNOPSIS

```
#include <pthread.h>

int pthread_mutexattr_getrobust(const pthread_mutexattr_t *restrict
    attr, int *restrict robust);
int pthread_mutexattr_setrobust(pthread_mutexattr_t *attr,
    int robust);
```

DESCRIPTION

The *pthread_mutexattr_getrobust*( ) and *pthread_mutexattr_setrobust*( ) functions, respectively, shall get and set the mutex *robust* attribute. This attribute is set in the *robust* parameter. Valid values for *robust* include:

PTHREAD_MUTEX_STALLED
: No special actions are taken if the owner of the mutex is terminated while holding the mutex lock. This can lead to deadlocks if no other thread can unlock the mutex.
This is the default value.

PTHREAD_MUTEX_ROBUST
: If the process containing the owning thread of a robust mutex terminates while holding the mutex lock, the next thread that acquires the mutex shall be notified about the termination by the return value [EOWNERDEAD] from the locking function. If the owning thread of a robust mutex terminates while holding the mutex lock, the next thread that acquires the mutex may be notified about the termination by the return value [EOWNERDEAD]. The notified thread can then attempt to mark the state protected by the mutex as consistent again by a call to *pthread_mutex_consistent*( ). After a subsequent successful call to *pthread_mutex_unlock*( ), the mutex lock shall be released and can be used normally by other threads. If the mutex is unlocked without a call to *pthread_mutex_consistent*( ), it shall be in a permanently unusable state and all attempts to lock the mutex shall fail with the error [ENOTRECOVERABLE]. The only permissible operation on such a mutex is *pthread_mutex_destroy*( ).

RETURN VALUE

Upon successful completion, the *pthread_mutexattr_getrobust*( ) function shall return zero and store the value of the *robust* attribute of *attr* into the object referenced by the *robust* parameter. Otherwise, an error value shall be returned to indicate the error. If successful, the *pthread_mutexattr_setrobust*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_mutexattr_setrobust*( ) function shall fail if:

[EINVAL] The value of *robust* is invalid.

The *pthread_mutexattr_getrobust*( ) and *pthread_mutexattr_setrobust*( ) functions may fail if:

[EINVAL] The value specified by *attr* is invalid.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

The actions required to make the state protected by the mutex consistent again are solely dependent on the application. If it is not possible to make the state of a mutex consistent, robust mutexes can be used to notify this situation by calling *pthread_mutex_unlock*( ) without a prior call to *pthread_mutex_consistent*( ).

If the state is declared inconsistent by calling *pthread_mutex_unlock*( ) without a prior call to *pthread_mutex_consistent*( ), a possible approach could be to destroy the mutex and then reinitialize it. However, it should be noted that this is possible only in certain situations where the state protected by the mutex has to be reinitialized and coordination achieved with other threads blocked on the mutex, because otherwise a call to a locking function with a reference to a mutex object invalidated by a call to *pthread_mutex_destroy*( ) results in undefined behavior.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_mutex_consistent( ), *pthread_mutex_destroy*( ), *pthread_mutex_lock*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 7.

The *pthread_mutexattr_getrobust*( ) and *pthread_mutexattr_setrobust*( ) functions are moved from the Threads option to the Base.

pthread_mutexattr_gettype( )

NAME

pthread_mutexattr_gettype, pthread_mutexattr_settype — get and set the mutex type attribute

SYNOPSIS

```
#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 thr

36726 **ERRORS**

36727 The *pthread_mutexattr_settype*( ) function shall fail if:

36728 [EINVAL] The value *type* is invalid.

36729 The *pthread_mutexattr_gettype*( ) and *pthread_mutexattr_settype*( ) functions may fail if:

36730 [EINVAL] The value specified by *attr* is invalid.

36731 These functions shall not return an error code of [EINTR].

36732 **EXAMPLES**

36733 None.

36734 **APPLICATION USAGE**

36735 It is advised that an application should not use a PTHREAD_MUTEX_RECURSIVE mutex with
36736 condition variables because the implicit unlock performed for a *pthread_cond_timedwait*( ) or
36737 *pthread_cond_wait*( ) may not actually release the mutex (if it had been locked multiple times). If
36738 this happens, no other thread can satisfy the condition of the predicate.

36739 **RATIONALE**

36740 None.

36741 **FUTURE DIRECTIONS**

36742 None.

36743 **SEE ALSO**

36744 *pthread_cond_timedwait( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

36745 **CHANGE HISTORY**

36746 First released in Issue 5.

36747 **Issue 6**

36748 The Open Group Corrigendum U033/3 is applied. The SYNOPSIS for
36749 *pthread_mutexattr_gettype*( ) is updated so that the first argument is of type **const**
36750 **pthread_mutexattr_t *.**

36751 The **restrict** keyword is added to the *pthread_mutexattr_gettype*( ) prototype for alignment with
36752 the ISO/IEC 9899: 1999 standard.

36753 **Issue 7**

36754 The *pthread_mutexattr_gettype*( ) and *pthread_mutexattr_settype*( ) functions are moved from the
36755 XSI option to the Base.

36756 **NAME**

36757 pthread_mutexattr_init — initialize the mutex attributes object

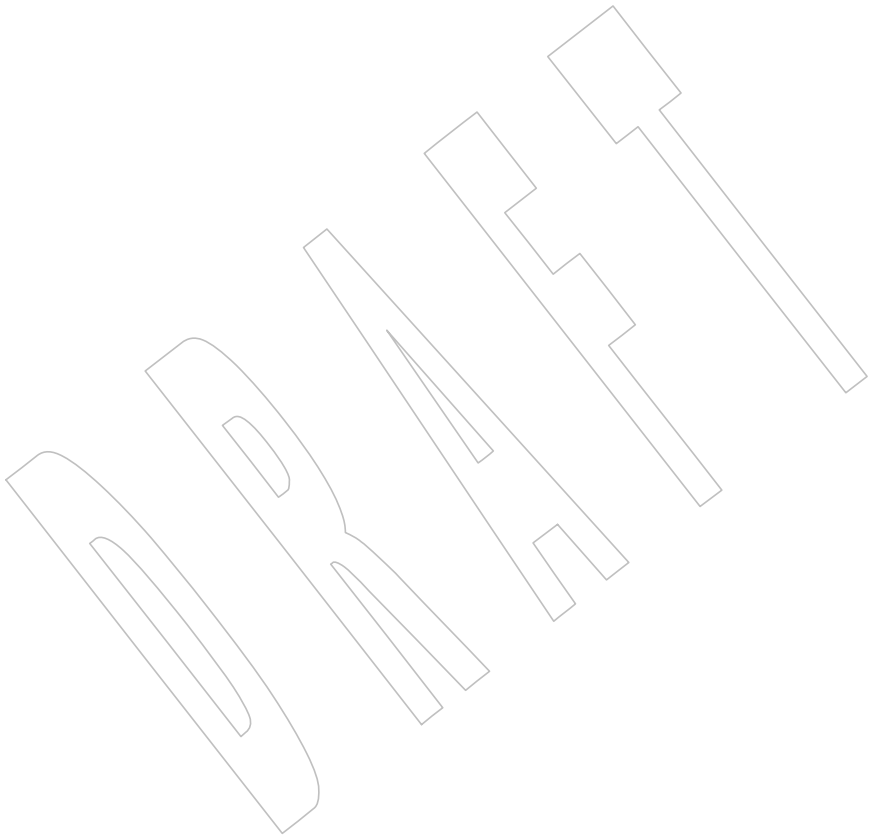
36758 **SYNOPSIS**

36759 `#include <pthread.h>`

36760 `int pthread_mutexattr_init(pthread_mutexattr_t *attr);`

36761 **DESCRIPTION**

36762 Refer to *pthread_mutexattr_destroy( )*.



36763 **NAME**

36764 pthread_mutexattr_setprioceiling — set the prioceiling attribute of the mutex attributes object
36765 (**REALTIME THREADS**)

36766 **SYNOPSIS**

36767 RPP | TPP
```
#include <pthread.h>

int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
    int prioceiling);
```

36770 **DESCRIPTION**

36771 Refer to *pthread_mutexattr_getprioceiling( )*.

NAME

pthread_mutexattr_setprotocol — set the protocol attribute of the mutex attributes object (**REALTIME THREADS**)

SYNOPSIS

MC1

```
#include <pthread.h>

int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
    int protocol);
```

DESCRIPTION

Refer to *pthread_mutexattr_getprotocol( )*.

36781 **NAME**

36782 pthread_mutexattr_setpshared — set the process-shared attribute

36783 **SYNOPSIS**

36784 TSH
```
#include <pthread.h>

int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
    int pshared);
```

36787 **DESCRIPTION**

36788 Refer to *pthread_mutexattr_getpshared( )*.

pthread_mutexattr_setrobust( )

36789 **NAME**

36790 pthread_mutexattr_setrobust — get and set the mutex robust attribute

36791 **SYNOPSIS**

36792
```
#include <pthread.h>
```

36793
```
int pthread_mutexattr_setrobust(pthread_mutexattr_t *attr,
```
36794
```
    int robust);
```

36795 **DESCRIPTION**

36796 Refer to *pthread_mutexattr_getrobust( )*.

36797 **NAME**

36798 pthread_mutexattr_settype — set the mutex type attribute

36799 **SYNOPSIS**

36800
```
#include <pthread.h>
```

36801
```
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

36802 **DESCRIPTION**

36803 Refer to *pthread_mutexattr_gettype( )*.

NAME

pthread_once — dynamic package initialization

SYNOPSIS

```
#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 cancellation point. However, if *init_routine* is a cancellation 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.

The *pthread_once*( ) function shall not return an error code of [EINTR].

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:

```
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:

```
#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 1003.1-200x, **<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.

Issue 7

The *pthread_once*( ) function is moved from the Threads option to the Base.

NAME

pthread_rwlock_destroy, pthread_rwlock_init — destroy and initialize a read-write lock object

SYNOPSIS

```
#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);
XSI pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
```

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.

XSI In cases where default read-write lock attributes are appropriate, the macro PTHREAD_RWLOCK_INITIALIZER can be used to initialize read-write locks that are statically allocated. The effect shall be equivalent to dynamic initialization by a call to *pthread_rwlock_init*( ) with the *attr* parameter specified as NULL, except that no error checks are performed.

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 [EINTR].

EXAMPLES

None.

APPLICATION USAGE

Applications using these and related read-write lock functions may be subject to priority inversion, as discussed in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

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 1003.1-200x, **<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.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/45 is applied, adding APPLICATION USAGE relating to priority inversion.

Issue 7

Austin Group Interpretation 1003.1-2001 #048 is applied, adding the PTHREAD_RWLOCK_INITIALIZER macro.

The *pthread_rwlock_destroy*( ) and *pthread_rwlock_init*( ) functions are moved from the Threads option to the Base.

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 Thread 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.

ERRORS

The *pthread_rwlock_tryrdlock*( ) 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] A deadlock condition was detected or 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 1003.1-200x, 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 1003.1-200x, Section 4.10, Memory Synchronization, **<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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/101 is applied, updating the ERRORS section so that the [EDEADLK] error includes detection of a deadlock condition.

Issue 7

The *pthread_rwlock_rdlock*( ) and *pthread_rwlock_tryrdlock*( ) functions are moved from the Threads option to the Base.

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.

The timeout shall be based on the CLOCK_REALTIME clock. The resolution of the timeout shall be the resolution of the CLOCK_REALTIME clock. 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.

The *pthread_rwlock_timedrdlock*( ) function may fail if:

[EAGAIN] The read lock could not be acquired because the maximum number of read locks for lock would be exceeded.

[EDEADLK] A deadlock condition was detected or 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.

This function shall not return an error code of [EINTR].

37102 **EXAMPLES**

37103 None.

37104 **APPLICATION USAGE**

37105 Applications using this function may be subject to priority inversion, as discussed in the Base
37106 Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

37107 **RATIONALE**

37108 None.

37109 **FUTURE DIRECTIONS**

37110 None.

37111 **SEE ALSO**

37112 *pthread_rwlock_destroy( )*, *pthread_rwlock_rdlock( )*, *pthread_rwlock_timedwrlock( )*,
37113 *pthread_rwlock_tryrdlock( )*, *pthread_rwlock_trywrlock( )*, *pthread_rwlock_unlock( )*,
37114 *pthread_rwlock_wrlock( )*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.10,
37115 Memory Synchronization, **<pthread.h>**, **<time.h>**

37116 **CHANGE HISTORY**

37117 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

37118 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/102 is applied, updating the ERRORS
37119 section so that the [EDEADLK] error includes detection of a deadlock condition.

37120 **Issue 7**

37121 The *pthread_rwlock_timedrdlock( )* function is moved from the Timeouts option to the Base.

NAME

pthread_rwlock_timedwrlock — lock a read-write lock for writing

SYNOPSIS

```
#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.

The timeout shall be based on the CLOCK_REALTIME clock. The resolution of the timeout shall be the resolution of the CLOCK_REALTIME clock. 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] A deadlock condition was detected or 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].

37161 **EXAMPLES**

37162 None.

37163 **APPLICATION USAGE**

37164 Applications using this function may be subject to priority inversion, as discussed in the Base
37165 Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

37166 **RATIONALE**

37167 None.

37168 **FUTURE DIRECTIONS**

37169 None.

37170 **SEE ALSO**

37171 *pthread_rwlock_destroy( )*, *pthread_rwlock_rdlock( )*, *pthread_rwlock_timedrdlock( )*,
37172 *pthread_rwlock_tryrdlock( )*, *pthread_rwlock_trywrlock( )*, *pthread_rwlock_unlock( )*,
37173 *pthread_rwlock_wrlock( )*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.10,
37174 Memory Synchronization, **<pthread.h>**, **<time.h>**

37175 **CHANGE HISTORY**

37176 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

37177 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/103 is applied, updating the ERRORS
37178 section so that the [EDEADLK] error includes detection of a deadlock condition.

37179 **Issue 7**

37180 The *pthread_rwlock_timedwrlock( )* function is moved from the Timeouts option to the Base.

pthread_rwlock_tryrdlock( )

37181 **NAME**

37182 pthread_rwlock_tryrdlock — lock a read-write lock object for reading

37183 **SYNOPSIS**

37184 ```
#include <pthread.h>
```

37185 ```
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```

37186 **DESCRIPTION**

37187 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:

EXAMPLES

None.

APPLICATION USAGE

Applications using these functions may be subject to priority inversion, as discussed in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_rwlock_destroy( ), *pthread_rwlock_rdlock*( ), *pthread_rwlock_timedrdlock*( ), *pthread_rwlock_timedwrlock*( ), *pthread_rwlock_tryrdlock*( ), *pthread_rwlock_unlock*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.10, Memory Synchronization, **<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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/104 is applied, updating the ERRORS section so that the [EDEADLK] error includes detection of a deadlock condition.

Issue 7

The *pthread_rwlock_trywrlock*( ) and *pthread_rwlock_wrlock*( ) functions are moved from the Threads option to the Base.

NAME

pthread_rwlock_unlock — unlock a read-write lock object

SYNOPSIS

```
#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 TPS 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 1003.1-200x, Section 4.10, Memory Synchronization, **<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:
  - — 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.

Issue 7

The *pthread_rwlock_unlock*( ) function is moved from the Threads option to the Base.

37310 **NAME**

37311 pthread_rwlock_wrlock — lock a read-write lock object for writing

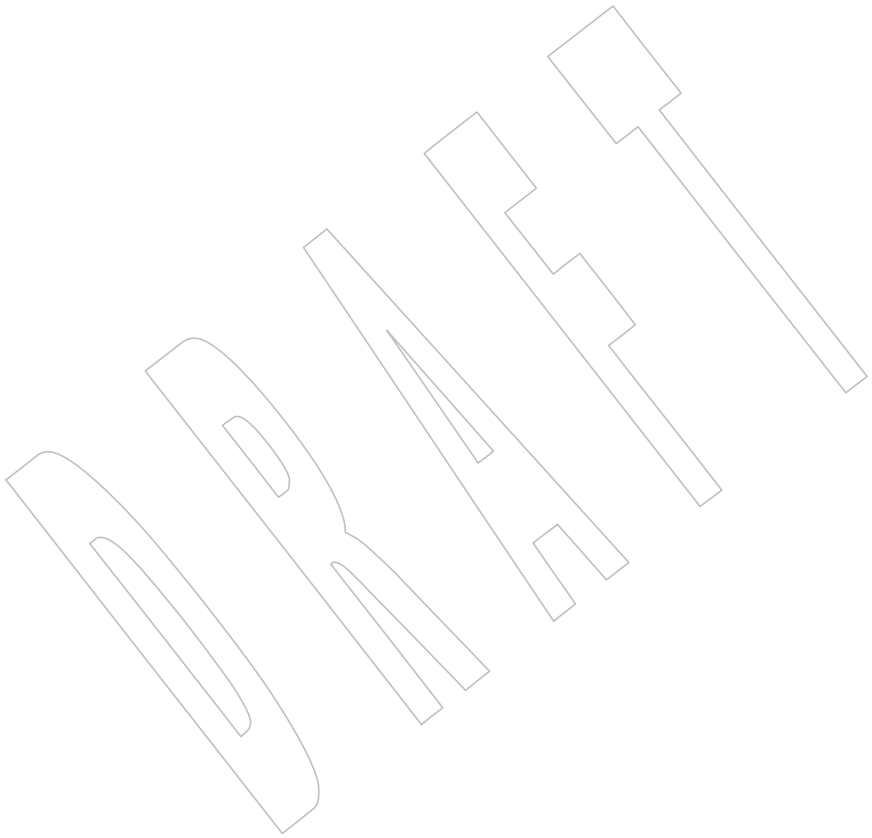
37312 **SYNOPSIS**

37313 `#include <pthread.h>`

37314 `int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);`

37315 **DESCRIPTION**

37316 Refer to *pthread_rwlock_trywrlock( )*.



NAME

pthread_rwlockattr_destroy, pthread_rwlockattr_init — destroy and initialize the read-write lock attributes object

SYNOPSIS

```
#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 1003.1-200x, **<pthread.h>**

37356 **CHANGE HISTORY**

37357 First released in Issue 5.

37358 **Issue 6**

37359 The following changes are made for alignment with IEEE Std 1003.1j-2000:

37360 37361 37362 • 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).

37363 • The SEE ALSO section is updated.

37364 **Issue 7**

37365 37366 The *pthread_rwlockattr_destroy*( ) and *pthread_rwlockattr_init*( ) functions are moved from the Threads option to the Base.

NAME

pthread_rwlockattr_getpshared, pthread_rwlockattr_setpshared — get and set the process-shared attribute of the read-write lock attributes object

SYNOPSIS

TSH
```
#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].

37403 **EXAMPLES**

37404 None.

37405 **APPLICATION USAGE**

37406 None.

37407 **RATIONALE**

37408 None.

37409 **FUTURE DIRECTIONS**

37410 None.

37411 **SEE ALSO**

37412 *pthread_rwlock_destroy( )*, *pthread_rwlockattr_destroy( )*, *pthread_rwlockattr_init( )*, the Base
37413 Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

37414 **CHANGE HISTORY**

37415 First released in Issue 5.

37416 **Issue 6**

37417 The following changes are made for alignment with IEEE Std 1003.1j-2000:

37418 • The margin code in the SYNOPSIS is changed to THR TSH to indicate that the
37419 functionality is now part of the Threads option (previously it was part of the Read-Write
37420 Locks option in IEEE Std 1003.1j-2000 and also part of the XSI extension).

37421 • The DESCRIPTION notes that additional attributes are implementation-defined.

37422 • The SEE ALSO section is updated.

37423 The **restrict** keyword is added to the *pthread_rwlockattr_getpshared( )* prototype for alignment
37424 with the ISO/IEC 9899: 1999 standard.

37425 **Issue 7**

37426 The *pthread_rwlockattr_getpshared( )* and *pthread_rwlockattr_setpshared( )* functions are moved from
37427 the Threads option.

NAME

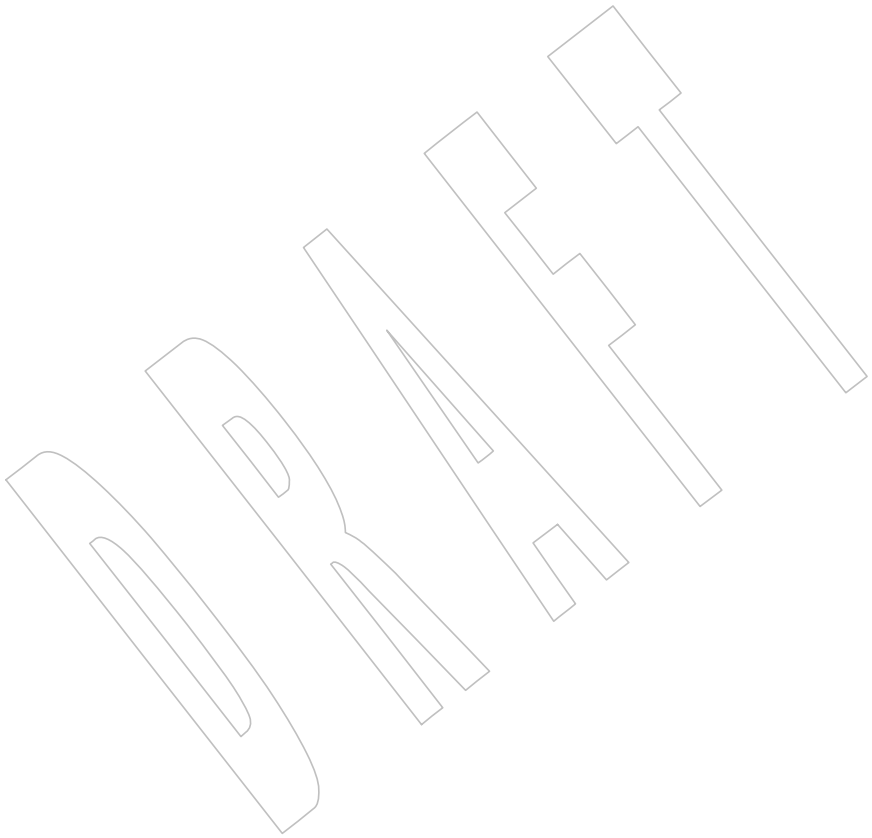
pthread_rwlockattr_init — initialize the read-write lock attributes object

SYNOPSIS

```
#include <pthread.h>

int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
```

DESCRIPTION

Refer to *pthread_rwlockattr_destroy( )*.



37435 **NAME**

37436 pthread_rwlockattr_setpshared — set the process-shared attribute of the read-write lock
37437 attributes object

37438 **SYNOPSIS**

37439 TSH
```
#include <pthread.h>

int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
    int pshared);
```

37442 **DESCRIPTION**

37443 Refer to *pthread_rwlockattr_getpshared( )*.

NAME

pthread_self — get the calling thread ID

SYNOPSIS

```
#include <pthread.h>

pthread_t pthread_self(void);
```

DESCRIPTION

The *pthread_self*( ) function shall return the thread ID of the calling thread.

RETURN VALUE

The *pthread_self*( ) 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

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 1003.1-200x, **<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.

Issue 7

Austin Group Interpretation 1003.1-2001 #063 is applied, updating the RETURN VALUE section.

The *pthread_self*( ) function is moved from the Threads option to the Base.

NAME

pthread_setcancelstate, pthread_setcanceltype, pthread_testcancel — set cancelability state

SYNOPSIS

```
#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 cancellation 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 cancellation 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 cancellation 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 1003.1-200x, **<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.

Issue 7

The *pthread_setcancelstate*( ), *pthread_setcanceltype*( ), and *pthread_testcancel*( ) functions are moved from the Threads option to the Base.

37544 **NAME**

37545 pthread_setconcurrency — set the level of concurrency

37546 **SYNOPSIS**

37547 OB XSI
```
#include <pthread.h>

int pthread_setconcurrency(int new_level);
```
37548

37549 **DESCRIPTION**

37550 Refer to *pthread_getconcurrency( )*.

NAME

pthread_setschedparam — dynamic thread scheduling parameters access (**REALTIME THREADS**)

SYNOPSIS

TPS

```
#include <pthread.h>

int pthread_setschedparam(pthread_t thread, int policy,
    const struct sched_param *param);
```

DESCRIPTION

Refer to *pthread_getschedparam( )*.

NAME

pthread_setschedprio — dynamic thread scheduling parameters access (**REALTIME THREADS**)

SYNOPSIS

TPS
```
#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 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 priority of the specified thread.

[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.

37600 **FUTURE DIRECTIONS**

37601 None.

37602 **SEE ALSO**

37603 Scheduling Policies (on page 44), *pthread_getschedparam*( ), the Base Definitions volume of
37604 IEEE Std 1003.1-200x, **<pthread.h>**

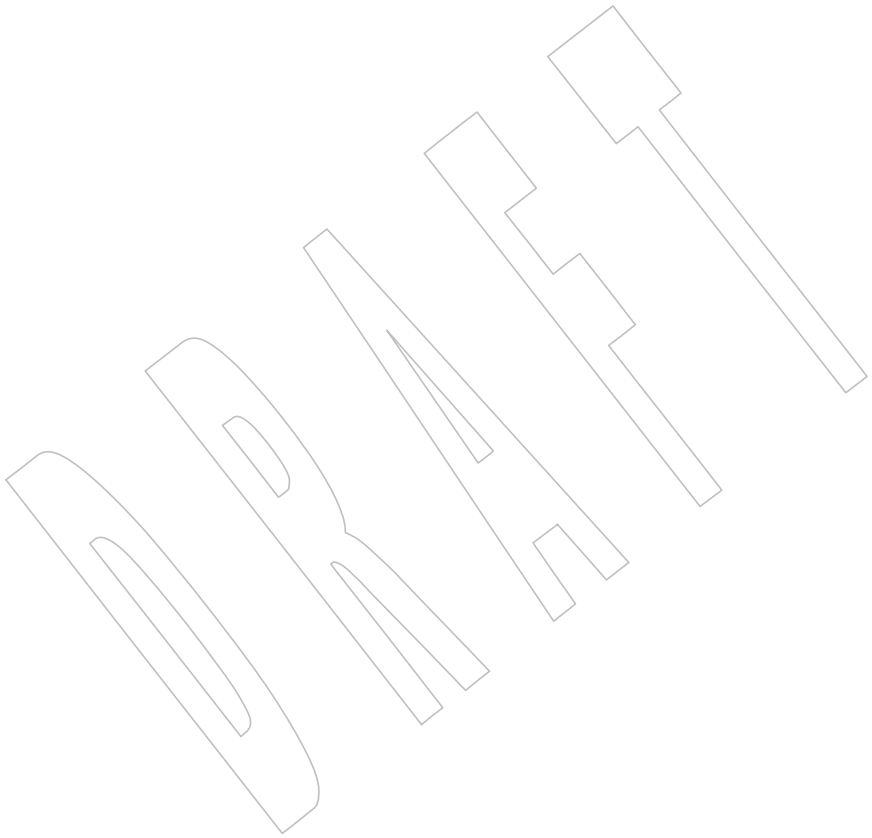
37605 **CHANGE HISTORY**

37606 First released in Issue 6. Included as a response to IEEE PASC Interpretation 1003.1 #96.

37607 **Issue 7**

37608 Austin Group Interpretation 1003.1-2001 #069 is applied, updating the [EPERM] error.

37609 The *pthread_setschedprio*( ) function is moved from the Threads option.



37610 **NAME**

37611 pthread_setspecific — thread-specific data management

37612 **SYNOPSIS**

37613
```
#include <pthread.h>
```

37614
```
int pthread_setspecific(pthread_key_t key, const void *value);
```

37615 **DESCRIPTION**

37616 Refer to *pthread_getspecific( )*.

NAME

pthread_sigmask, sigprocmask — examine and change blocked signals

SYNOPSIS

CX
```
#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 thread's 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 signal mask of the process shall be unchanged.

ERRORS

The *pthread_sigmask*( ) and *sigprocmask*( ) functions 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].

EXAMPLES

Signalling in a Multi-Threaded Process

This example shows the use of *pthread_sigmask*( ) in order to deal with signals in a multi-threaded process. It provides a fairly general framework that could be easily adapted/extended.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <signal.h>
#include <string.h>
#include <errno.h>
...

static sigset_t   signal_mask;  /* signals to block         */

int main (int argc, char *argv[])
{
    pthread_t  sig_thr_id;      /* signal handler thread ID */
    int        rc;              /* return code              */

    sigemptyset (&signal_mask);
    sigaddset (&signal_mask, SIGINT);
    sigaddset (&signal_mask, SIGTERM);
    rc = pthread_sigmask (SIG_BLOCK, &signal_mask, NULL);
    if (rc != 0) {
        /* handle error */
        ...
    }
    /* any newly created threads inherit the signal mask */

    rc = pthread_create (&sig_thr_id, NULL, signal_thread, NULL);
    if (rc != 0) {
        /* handle error */
        ...
    }

    /* APPLICATION CODE */
    ...
}

void *signal_thread (void *arg)
{
    int       sig_caught;    /* signal caught       */
    int       rc;            /* returned code       */

    rc = sigwait (&signal_mask, &sig_caught);
    if (rc != 0) {
        /* handle error */
    }
    switch (sig_caught)
    {
```



```
        case SIGINT:     /* process SIGINT  */
            ...
            break;
        case SIGTERM:    /* process SIGTERM */
            ...
            break;
        default:         /* should normally not happen */
            fprintf (stderr, "\nUnexpected signal %d\n", sig_caught);
            break;
        }
    }
```

APPLICATION USAGE

None.

RATIONALE

When a thread's 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.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *sigaction*( ), *sigaddset*( ), *sigdelset*( ), *sigemptyset*( ), *sigfillset*( ), *sigismember*( ), *sigpending*( ), *sigqueue*( ), *sigsuspend*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.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.

The *pthread_sigmask*( ) function is added for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_sigmask*( ) function is marked as part of the Threads option.

The SYNOPSIS 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 normative text 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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/105 is applied, updating "process' signal mask" to "thread's signal mask" in the DESCRIPTION and RATIONALE sections.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/106 is applied, adding the example to the EXAMPLES section.

Issue 7

The *pthread_sigmask*( ) function is moved from the Threads option to the Base.

pthread_spin_destroy( )

NAME

pthread_spin_destroy, pthread_spin_init — destroy or initialize a spin lock object

SYNOPSIS

```
#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

[ENOMEM] Insufficient memory exists to initialize the lock.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_spin_lock( ), *pthread_spin_unlock( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

Issue 7

The *pthread_spin_destroy*( ) and *pthread_spin_init*( ) functions are moved from the Spin Locks option to the Base.

NAME

pthread_spin_lock, pthread_spin_trylock — lock a spin lock object

SYNOPSIS

```
#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.

The *pthread_spin_lock*( ) function may fail if:

[EDEADLK] A deadlock condition was detected or the calling thread already holds the lock.

The *pthread_spin_trylock*( ) function shall fail if:

[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 1003.1-200x, Section 3.285, Priority Inversion.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_spin_destroy( ), *pthread_spin_unlock*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.10, Memory Synchronization, **<pthread.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/107 is applied, updating the ERRORS

37852 section so that the [EDEADLK] error includes detection of a deadlock condition.

37853 **Issue 7**

37854 The *pthread_spin_lock*( ) and *pthread_spin_trylock*( ) functions are moved from the Spin Locks
37855 option to the Base.

NAME

pthread_spin_unlock — unlock a spin lock object

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

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_spin_destroy( ), *pthread_spin_lock*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.10, Memory Synchronization, **<pthread.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

Issue 7

The *pthread_spin_unlock*( ) function is moved from the Spin Locks option to the Base.

37892 **NAME**

37893 pthread_testcancel — set cancelability state

37894 **SYNOPSIS**

37895
```
#include <pthread.h>
```

37896
```
void pthread_testcancel(void);
```

37897 **DESCRIPTION**

37898 Refer to *pthread_setcancelstate( )*.

NAME

ptsname — get name of the slave pseudo-terminal device

SYNOPSIS

XSI

```
#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 thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

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 1003.1-200x, **<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.

37934 **NAME**

37935 putc — put a byte on a stream

37936 **SYNOPSIS**

37937
```
#include <stdio.h>
```

37938
```
int putc(int c, FILE *stream);
```

37939 **DESCRIPTION**

37940 CX The functionality described on this reference page is aligned with the ISO C standard. Any
37941 conflict between the requirements described here and the ISO C standard is unintentional. This
37942 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

37943 The *putc*( ) function shall be equivalent to *fputc*( ), except that if it is implemented as a macro it
37944 may evaluate *stream* more than once, so the argument should never be an expression with side
37945 effects.

37946 **RETURN VALUE**

37947 Refer to *fputc*( ).

37948 **ERRORS**

37949 Refer to *fputc*( ).

37950 **EXAMPLES**

37951 None.

37952 **APPLICATION USAGE**

37953 Since it may be implemented as a macro, *putc*( ) may treat a *stream* argument with side effects
37954 incorrectly. In particular, *putc*(*c*,*f*++) does not necessarily work correctly. Therefore, use of this
37955 function is not recommended in such situations; *fputc*( ) should be used instead.

37956 **RATIONALE**

37957 None.

37958 **FUTURE DIRECTIONS**

37959 None.

37960 **SEE ALSO**

37961 *fputc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

37962 **CHANGE HISTORY**

37963 First released in Issue 1. Derived from Issue 1 of the SVID.

37964 **NAME**

37965 putc_unlocked — stdio with explicit client locking

37966 **SYNOPSIS**

37967 CX
```
#include <stdio.h>

int putc_unlocked(int c, FILE *stream);
```

37969 **DESCRIPTION**

37970 Refer to *getc_unlocked( )*.

37971 **NAME**

37972 putchar — put a byte on a stdout stream

37973 **SYNOPSIS**

37974
```
#include <stdio.h>
```

37975
```
int putchar(int c);
```

37976 **DESCRIPTION**

37977 CX The functionality described on this reference page is aligned with the ISO C standard. Any
37978 conflict between the requirements described here and the ISO C standard is unintentional. This
37979 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

37980 The function call *putchar*(*c*) shall be equivalent to *putc*(*c*,*stdout*).

37981 **RETURN VALUE**

37982 Refer to *fputc*( ).

37983 **ERRORS**

37984 Refer to *fputc*( ).

37985 **EXAMPLES**

37986 None.

37987 **APPLICATION USAGE**

37988 None.

37989 **RATIONALE**

37990 None.

37991 **FUTURE DIRECTIONS**

37992 None.

37993 **SEE ALSO**

37994 *putc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

37995 **CHANGE HISTORY**

37996 First released in Issue 1. Derived from Issue 1 of the SVID.

putchar_unlocked( )

NAME

putchar_unlocked — stdio with explicit client locking

SYNOPSIS

CX

```
#include <stdio.h>

int putchar_unlocked(int c);
```

DESCRIPTION

Refer to *getc_unlocked( )*.

38004 **NAME**

38005 putenv — change or add a value to an environment

38006 **SYNOPSIS**

38007 XSI
```
#include <stdlib.h>

int putenv(char *string);
```

38009 **DESCRIPTION**

38010 The *putenv*( ) function shall use the *string* argument to set environment variable values. The
38011 *string* argument should point to a string of the form "*name=value*". The *putenv*( ) function shall
38012 make the value of the environment variable *name* equal to *value* by altering an existing variable
38013 or creating a new one. In either case, the string pointed to by *string* shall become part of the
38014 environment, so altering the string shall change the environment. The space used by *string* is no
38015 longer used once a new string which defines *name* is passed to *putenv*( ).

38016 The *putenv*( ) function need not be thread-safe. A function that is not required to be thread-safe is
38017 not required to be reentrant.

38018 **RETURN VALUE**

38019 Upon successful completion, *putenv*( ) shall return 0; otherwise, it shall return a non-zero value
38020 and set *errno* to indicate the error.

38021 **ERRORS**

38022 The *putenv*( ) function may fail if:

38023 [ENOMEM] Insufficient memory was available.

38024 **EXAMPLES**

38025 **Changing the Value of an Environment Variable**

38026 The following example changes the value of the *HOME* environment variable to the value
38027 **/usr/home**.

```
38028 #include <stdlib.h>
38029 ...
38030 static char *var = "HOME=/usr/home";
38031 int ret;

38032 ret = putenv(var);
```

38033 **APPLICATION USAGE**

38034 The *putenv*( ) function manipulates the environment pointed to by *environ*, and can be used in
38035 conjunction with *getenv*( ).

38036 See *exec*, for restrictions on changing the environment in multi-threaded applications.

38037 This routine may use *malloc*( ) to enlarge the environment.

38038 A potential error is to call *putenv*( ) with an automatic variable as the argument, then return from
38039 the calling function while *string* is still part of the environment.

38040 The *setenv*( ) function is preferred over this function.

38041 **RATIONALE**

38042 The standard developers noted that *putenv*( ) is the only function available to add to the
38043 environment without permitting memory leaks.

38044 **FUTURE DIRECTIONS**

38045 None.

38046 **SEE ALSO**

38047 *exec*, *getenv*( ), *malloc*( ), *setenv*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

38048 **CHANGE HISTORY**

38049 First released in Issue 1. Derived from Issue 1 of the SVID.

38050 **Issue 5**

38051 The type of the argument to this function is changed from **const char** * to **char** *. This was
38052 indicated as a FUTURE DIRECTION in previous issues.

38053 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

38054 **Issue 6**

38055 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/48 is applied, clarifying wording in the
38056 DESCRIPTION and adding a new paragraph into APPLICATION USAGE referring readers to
38057 *exec*.

NAME

putmsg, putpmsg — send a message on a STREAM (**STREAMS**)

SYNOPSIS

OB XSR

```
#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 *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*.

```
#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);
```

Using putpmsg( )

This example has the same effect as the previous example. In this example, however, the *putpmsg*( ) function creates and sends the message to the STREAMS file.

```
#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);
```

APPLICATION USAGE

None.

RATIONALE

None.

38181 **FUTURE DIRECTIONS**

38182 The *putmsg*( ) and *putpmsg*( ) functions may be removed in a future version.

38183 **SEE ALSO**

38184 Section 2.6 (on page 38), *getmsg*( ), *poll*( ), *read*( ), *write*( ), the Base Definitions volume of
38185 IEEE Std 1003.1-200x, **<stropts.h>**

38186 **CHANGE HISTORY**

38187 First released in Issue 4, Version 2.

38188 **Issue 5**

38189 Moved from X/OPEN UNIX extension to BASE.

38190 The following text is removed from the DESCRIPTION: "The STREAM head guarantees that the
38191 control part of a message generated by *putmsg*( ) is at least 64 bytes in length".

38192 **Issue 6**

38193 This function is marked as part of the XSI STREAMS Option Group.

38194 The normative text is updated to avoid use of the term "must" for application requirements.

38195 **Issue 7**

38196 The *putmsg*( ) and *putpmsg*( ) functions are marked obsolescent.

38197 **NAME**

38198 puts — put a string on standard output

38199 **SYNOPSIS**

38200
```
#include <stdio.h>
```

38201
```
int puts(const char *s);
```

38202 **DESCRIPTION**

38203 CX The functionality described on this reference page is aligned with the ISO C standard. Any
38204 conflict between the requirements described here and the ISO C standard is unintentional. This
38205 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

38206 The *puts*( ) function shall write the string pointed to by *s*, followed by a <newline>, to the
38207 standard output stream *stdout*. The terminating null byte shall not be written.

38208 CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful
38209 execution of *puts*( ) and the next successful completion of a call to *fflush*( ) or *fclose*( ) on the same
38210 stream or a call to *exit*( ) or *abort*( ).

38211 **RETURN VALUE**

38212 Upon successful completion, *puts*( ) shall return a non-negative number. Otherwise, it shall
38213 CX return EOF, shall set an error indicator for the stream, and *errno* shall be set to indicate the error.

38214 **ERRORS**

38215 Refer to *fputc*( ).

38216 **EXAMPLES**

38217 **Printing to Standard Output**

38218 The following example gets the current time, converts it to a string using *localtime*( ) and
38219 *asctime*( ), and prints it to standard output using *puts*( ). It then prints the number of minutes to
38220 an event for which it is waiting.

38221
```
#include <time.h>
```
38222
```
#include <stdio.h>
```
38223
```
...
```
38224
```
time_t now;
```
38225
```
int minutes_to_event;
```
38226
```
...
```
38227
```
time(&now);
```
38228
```
printf("The time is ");
```
38229
```
puts(asctime(localtime(&now)));
```
38230
```
printf("There are %d minutes to the event.\n",
```
38231
```
    minutes_to_event);
```
38232
```
...
```

38233 **APPLICATION USAGE**

38234 The *puts*( ) function appends a <newline>, while *fputs*( ) does not.

38235 **RATIONALE**

38236 None.

38237 **FUTURE DIRECTIONS**

38238 None.

38239 **SEE ALSO**

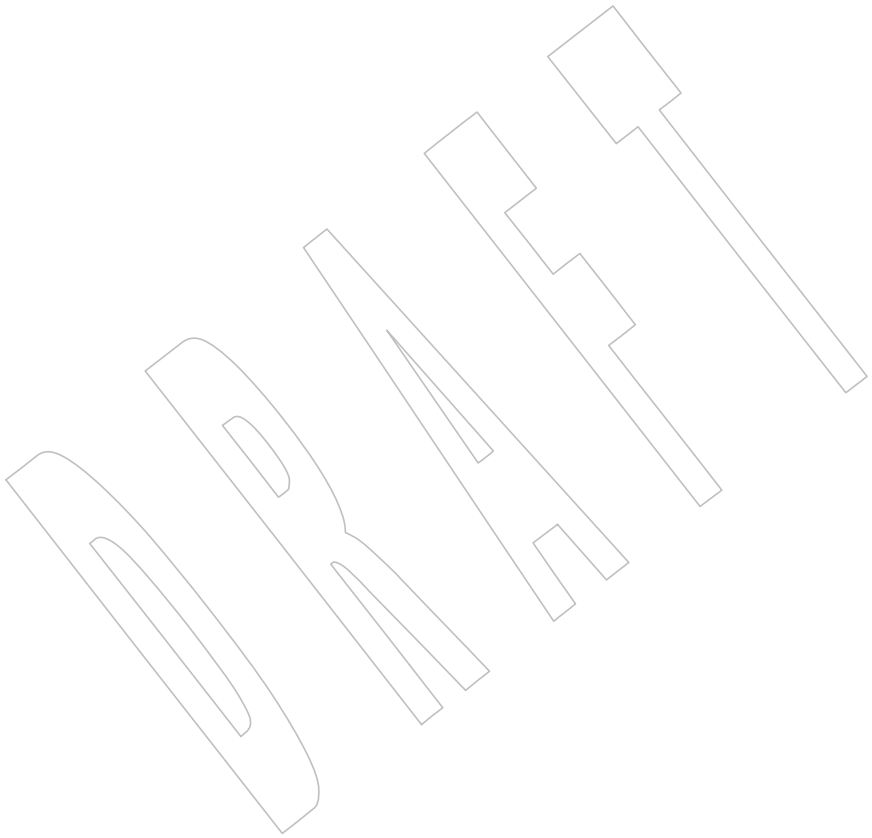
38240 *fopen*( ), *fputs*( ), *putc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

38241 **CHANGE HISTORY**

38242 First released in Issue 1. Derived from Issue 1 of the SVID.

38243 **Issue 6**

38244 Extensions beyond the ISO C standard are marked.



38245 **NAME**

38246 pututxline — put an entry into the user accounting database

38247 **SYNOPSIS**

38248 XSI
```
#include <utmpx.h>

struct utmpx *pututxline(const struct utmpx *utmpx);
```
38249

38250 **DESCRIPTION**

38251 Refer to *endutxent( )*.

38252 **NAME**

38253 putwc — put a wide character on a stream

38254 **SYNOPSIS**

38255 ```
#include <stdio.h>
```

38256 ```
#include <wchar.h>
```

38257 ```
wint_t putwc(wchar_t wc, FILE *stream);
```

38258 **DESCRIPTION**

38259 CX The functionality described on this reference page is aligned with the ISO C standard. Any
38260 conflict between the requirements described here and the ISO C standard is unintentional. This
38261 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

38262 The *putwc*( ) function shall be equivalent to *fputwc*( ), except that if it is implemented as a macro
38263 it may evaluate *stream* more than once, so the argument should never be an expression with side
38264 effects.

38265 **RETURN VALUE**

38266 Refer to *fputwc*( ).

38267 **ERRORS**

38268 Refer to *fputwc*( ).

38269 **EXAMPLES**

38270 None.

38271 **APPLICATION USAGE**

38272 Since it may be implemented as a macro, *putwc*( ) may treat a *stream* argument with side effects
38273 incorrectly. In particular, *putwc*(*wc*,*f*++) need not work correctly. Therefore, use of this function
38274 is not recommended; *fputwc*( ) should be used instead.

38275 **RATIONALE**

38276 None.

38277 **FUTURE DIRECTIONS**

38278 None.

38279 **SEE ALSO**

38280 *fputwc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**, **<wchar.h>**

38281 **CHANGE HISTORY**

38282 First released as a World-wide Portability Interface in Issue 4.

38283 **Issue 5**

38284 Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the type of argument *wc*
38285 is changed from **wint_t** to **wchar_t**.

38286 The Optional Header (OH) marking is removed from **<stdio.h>**.
```

38287 **NAME**

38288 putwchar — put a wide character on a stdout stream

38289 **SYNOPSIS**

38290
```
#include <wchar.h>
```

38291
```
wint_t putwchar(wchar_t wc);
```

38292 **DESCRIPTION**

38293 CX The functionality described on this reference page is aligned with the ISO C standard. Any
38294 conflict between the requirements described here and the ISO C standard is unintentional. This
38295 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

38296 The function call *putwchar*(*wc*) shall be equivalent to *putwc*(*wc*,*stdout*).

38297 **RETURN VALUE**

38298 Refer to *fputwc*( ).

38299 **ERRORS**

38300 Refer to *fputwc*( ).

38301 **EXAMPLES**

38302 None.

38303 **APPLICATION USAGE**

38304 None.

38305 **RATIONALE**

38306 None.

38307 **FUTURE DIRECTIONS**

38308 None.

38309 **SEE ALSO**

38310 *fputwc*( ), *putwc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

38311 **CHANGE HISTORY**

38312 First released in Issue 4.

38313 **Issue 5**

38314 Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the type of argument *wc*
38315 is changed from **wint_t** to **wchar_t**.

38316 **NAME**

38317 pwrite — write on a file

38318 **SYNOPSIS**

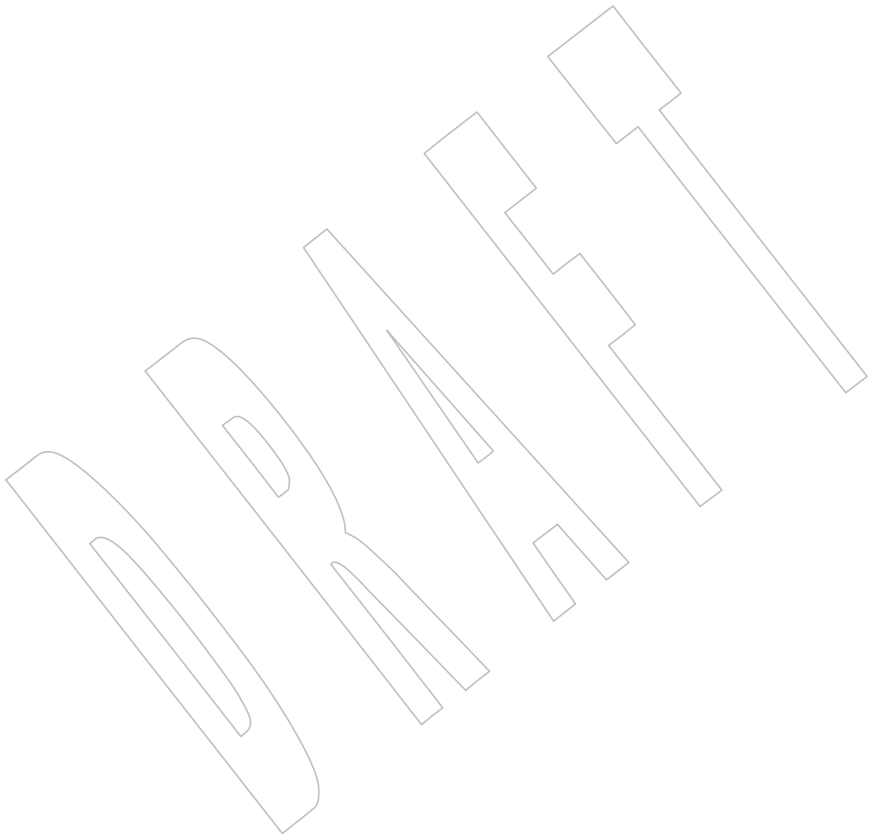
38319
```
#include <unistd.h>
```

38320
```
ssize_t pwrite(int fildes, const void *buf, size_t nbyte,
```

38321
```
    off_t offset);
```

38322 **DESCRIPTION**

38323 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

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 1003.1-200x defers to the ISO C standard.

The *qsort*( ) function shall sort an array of *nel* objects, the initial element of which is pointed to by *base*. The size of each object, in bytes, is specified by the *width* argument. If the *nel* argument has the value zero, the comparison function pointed to by *compar* shall not be called and no rearrangement shall take place.

The application shall ensure that the comparison function pointed to by *compar* does not alter the contents of the array. The implementation may reorder elements of the array between calls to the comparison function, but shall not alter the contents of any individual element.

When the same objects (consisting of width bytes, irrespective of their current positions in the array) are passed more than once to the comparison function, the results shall be consistent with one another. That is, they shall define a total ordering on the array.

The contents of the array shall be sorted in ascending order according to a comparison function. The *compar* argument is a pointer to the comparison function, which is called with two arguments that point to the elements being compared. The application shall ensure that the function returns an integer less than, equal to, or greater than 0, if the first argument is considered respectively less than, equal to, or greater than the second. If two members compare as equal, their order in the sorted array is unspecified.

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

The requirement that each argument (hereafter referred to as *p)* to the comparison function is a pointer to elements of the array implies that for every call, for each argument separately, all of the following expressions are non-zero:

```
((char *)p – (char *)base) % width == 0
(char *)p >= (char *)base
(char *)p < (char *)base + nel * width
```


FUTURE DIRECTIONS

None.

SEE ALSO

alphasort( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

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

Issue 6

The normative text is updated to avoid use of the term "must" for application requirements.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/49 is applied, adding the last sentence to the first non-shaded paragraph in the DESCRIPTION, and the following two paragraphs. The RATIONALE is also updated. These changes are for alignment with the ISO C standard.

38377 **NAME**

38378 raise — send a signal to the executing process

38379 **SYNOPSIS**

38380
```
#include <signal.h>
```

38381
```
int raise(int sig);
```

38382 **DESCRIPTION**

38383 CX The functionality described on this reference page is aligned with the ISO C standard. Any
38384 conflict between the requirements described here and the ISO C standard is unintentional. This
38385 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

38386 CX The *raise*( ) function shall send the signal *sig* to the executing thread or process. If a signal
38387 handler is called, the *raise*( ) function shall not return until after the signal handler does.

38388 CX The effect of the *raise*( ) function shall be equivalent to calling:

38389
```
pthread_kill(pthread_self(), sig);
```

38390 **RETURN VALUE**

38391 CX Upon successful completion, 0 shall be returned. Otherwise, a non-zero value shall be returned
38392 and *errno* shall be set to indicate the error.

38393 **ERRORS**

38394 The *raise*( ) function shall fail if:

38395 CX [EINVAL] The value of the *sig* argument is an invalid signal number.

38396 **EXAMPLES**

38397 None.

38398 **APPLICATION USAGE**

38399 None.

38400 **RATIONALE**

38401 The term "thread" is an extension to the ISO C standard.

38402 **FUTURE DIRECTIONS**

38403 None.

38404 **SEE ALSO**

38405 *kill*( ), *sigaction*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**,
38406 **<sys/types.h>**

38407 **CHANGE HISTORY**

38408 First released in Issue 4. Derived from the ANSI C standard.

38409 **Issue 5**

38410 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

38411 **Issue 6**

38412 Extensions beyond the ISO C standard are marked.

38413 The following new requirements on POSIX implementations derive from alignment with the
38414 Single UNIX Specification:

38415 • In the RETURN VALUE section, the requirement to set *errno* on error is added.

- The [EINVAL] error condition is added.

Issue 7

Functionality relating to the Threads option is moved to the Base.

NAME

rand, rand_r, srand — pseudo-random number generator

SYNOPSIS

```
#include <stdlib.h>

int rand(void);
CX int rand_r(unsigned *seed);
void srand(unsigned seed);
```

DESCRIPTION

CX For *rand*( ) and *srand*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *rand*( ) function shall compute a sequence of pseudo-random integers in the range [0,{RAND_MAX}] XSI with a period of at least 2^{32}.

CX The *rand*( ) function need not be thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

The *rand_r*( ) function shall compute a sequence of pseudo-random integers in the range [0,{RAND_MAX}]. (The value of the {RAND_MAX} macro shall be at least 32 767.)

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 1003.1-200x calls *rand*( ) or *srand*( ).

RETURN VALUE

The *rand*( ) function shall return the next pseudo-random number in the sequence.

CX 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 *drand48*( ) function provides a much more elaborate random number generator.

The limitations on the amount of state that can be carried between one function call and another mean the *rand_r*( ) function can never be implemented in a way which satisfies all of the requirements on a pseudo-random number generator. Therefore this function should be avoided whenever non-trivial requirements (including safety) have to be fulfilled.

RATIONALE

The ISO C standard *rand*( ) and *srand*( ) functions allow per-process pseudo-random streams shared by all threads. Those two functions need not change, but there has to be mutual-exclusion that prevents interference between two threads concurrently accessing the random number generator.

With regard to *rand*( ), there are two different behaviors that may be wanted in a multi-threaded program:

1. A single per-process sequence of pseudo-random numbers that is shared by all threads that call *rand*( )
2. A different sequence of pseudo-random numbers for each thread that calls *rand*( )

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.

38508 This does not address the known deficiencies of the *rand*( ) function implementations, which
38509 have been approached by maintaining more state. In effect, this specifies new thread-safe forms
38510 of a deficient function.

38511 **FUTURE DIRECTIONS**

38512 None.

38513 **SEE ALSO**

38514 *drand48*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

38515 **CHANGE HISTORY**

38516 First released in Issue 1. Derived from Issue 1 of the SVID.

38517 **Issue 5**

38518 The *rand_r*( ) function is included for alignment with the POSIX Threads Extension.

38519 A note indicating that the *rand*( ) function need not be reentrant is added to the DESCRIPTION.

38520 **Issue 6**

38521 Extensions beyond the ISO C standard are marked.

38522 The *rand_r*( ) function is marked as part of the Thread-Safe Functions option.

38523 **Issue 7**

38524 The *rand_r*( ) function is moved from the Thread-Safe Functions option to the Base.

38525 **NAME**

38526 random — generate pseudo-random number

38527 **SYNOPSIS**

38528 XSI
```
#include <stdlib.h>

long random(void);
```

38530 **DESCRIPTION**

38531 Refer to *initstate( )*.

NAME

pread, read — read from a file

SYNOPSIS

```
#include <unistd.h>

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

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 [EINTR].

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.

SIO 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.

SHM If *fildes* refers to a shared memory object, the result of the *read*( ) function is unspecified.

TYM If *fildes* refers to a typed memory object, the result of the *read*( ) function is unspecified.

OB XSR 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 I_SRDOPT *ioctl*( ) request, and can be tested with I_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 I_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 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 the STREAM head read queue is empty. Thereafter, it shall return 0.

The *pread*( ) function shall be equivalent to *read*( )

[ENOBUFS] 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*.

```
#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 1003.1-200x 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.

38703 **Input and Output**

38704 The use of I/O with large byte counts has always presented problems. Ideas such as *lread*( ) and
38705 *lwrite*( ) (using and returning **long**s) were considered at one time. The current solution is to use
38706 abstract types on the ISO C standard function to *read*( ) and *write*( ). The abstract types can be
38707 declared so that existing functions work, but can also be declared so that larger types can be
38708 represented in future implementations. It is presumed that whatever constraints limit the
38709 maximum range of **size_t** also limit portable I/O requests to the same range. This volume of
38710 IEEE Std 1003.1-200x also limits the range further by requiring that the byte count be limited so
38711 that a signed return value remains meaningful. Since the return type is also a (signed) abstract
38712 type, the byte count can be defined by the implementation to be larger than an **int** can hold.

38713 The standard developers considered adding atomicity requirements to a pipe or FIFO, but
38714 recognized that due to the nature of pipes and FIFOs there could be no guarantee of atomicity of
38715 reads of {PIPE_BUF} or any other size that would be an aid to applications portability.

38716 This volume of IEEE Std 1003.1-200x requires that no action be taken for *read*( ) or *write*( ) when
38717 *nbyte* is zero. This is not intended to take precedence over detection of errors (such as invalid
38718 buffer pointers or file descriptors). This is consistent with the rest of this volume of
38719 IEEE Std 1003.1-200x, but the phrasing here could be misread to require detection of the zero
38720 case before any other errors. A value of zero is to be considered a correct value, for which the
38721 semantics are a no-op.

38722 I/O is intended to be atomic to ordinary files and pipes and FIFOs. Atomic means that all the
38723 bytes from a single operation that started out together end up together, without interleaving
38724 from other I/O operations. It is a known attribute of terminals that this is not honored, and
38725 terminals are explicitly (and implicitly permanently) excepted, making the behavior unspecified.
38726 The behavior for other device types is also left unspecified, but the wording is intended to imply
38727 that future standards might choose to specify atomicity (or not).

38728 There were recommendations to add format parameters to *read*( ) and *write*( ) in order to handle
38729 networked transfers among heterogeneous file system and base hardware types. Such a facility
38730 may be required for support by the OSI presentation of layer services. However, it was
38731 determined that this should correspond with similar C-language facilities, and that is beyond
38732 the scope of this volume of IEEE Std 1003.1-200x. The concept was suggested to the developers
38733 of the ISO C standard for their consideration as a possible area for future work.

38734 In 4.3 BSD, a *read*( ) or *write*( ) that is interrupted by a signal before transferring any data does
38735 not by default return an [EINTR] error, but is restarted. In 4.2 BSD, 4.3 BSD, and the Eighth
38736 Edition, there is an additional function, *select*( ), whose purpose is to pause until specified
38737 activity (data to read, space to write, and so on) is detected on specified file descriptors. It is
38738 common in applications written for those systems for *select*( ) to be used before *read*( ) in
38739 situations (such as keyboard input) where interruption of I/O due to a signal is desired.

38740 The issue of which files or file types are interruptible is considered an implementation design
38741 issue. This is often affected primarily by hardware and reliability issues.

38742 There are no references to actions taken following an "unrecoverable error". It is considered
38743 beyond the scope of this volume of IEEE Std 1003.1-200x to describe what happens in the case of
38744 hardware errors.

38745 Previous versions of IEEE Std 1003.1-200x allowed two very different behaviors with regard to
38746 the handling of interrupts. In order to minimize the resulting confusion, it was decided that
38747 IEEE Std 1003.1-200x should support only one of these behaviors. Historical practice on
38748 AT&T-derived systems was to have *read*( ) and *write*( ) return –1 and set *errno* to [EINTR] when
38749 interrupted after some, but not all, of the data requested had been transferred. However, the U.S.
38750 Department of Commerce FIPS 151-1 and FIPS 151-2 require the historical BSD behavior, in
38751 which *read*( ) and *write*( ) return the number of bytes actually transferred before the interrupt. If
38752 –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 1003.1-200x is to return the number of bytes transferred.

IEEE Std 1003.1-200x 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 1003.1-200x, 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 made optional.

The following error conditions are added for operations on sockets: [ENOBUFS] and [ENOMEM].

The *readv*( ) function is split out into a separate reference page.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/108 is applied, updating the [EAGAIN] error in the ERRORS section from "the process would be delayed" to "the thread would be delayed".

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/109 is applied, making an editorial correction in the RATIONALE section.

Issue 7

The *pread*( ) function is moved from the XSI option to the Base.

Functionality relating to the XSI STREAMS option is marked obsolescent.

38812 **NAME**

38813 readdir, readdir_r — read a directory

38814 **SYNOPSIS**

38815
```
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
int readdir_r(DIR *restrict dirp, struct dirent *restrict entry,
    struct dirent **restrict result);
```

38819 **DESCRIPTION**

38820 The type **DIR**, which is defined in the **<dirent.h>** header, represents a *directory stream*, which is
38821 an ordered sequence of all the directory entries in a particular directory. Directory entries
38822 represent files; files may be removed from a directory or added to a directory asynchronously to
38823 the operation of *readdir*( ).

38824 The *readdir*( ) function shall return a pointer to a structure representing the directory entry at the
38825 current position in the directory stream specified by the argument *dirp*, and position the
38826 directory stream at the next entry. It shall return a null pointer upon reaching the end of the
38827 directory stream. The structure **dirent** defined in the **<dirent.h>** header describes a directory
38828 entry.

38829 The *readdir*( ) function shall not return directory entries containing empty names. If entries for
38830 dot or dot-dot exist, one entry shall be returned for dot and one entry shall be returned for dot-
38831 dot; otherwise, they shall not be returned.

38832 The pointer returned by *readdir*( ) points to data which may be overwritten by another call to
38833 *readdir*( ) on the same directory stream. This data is not overwritten by another call to *readdir*( )
38834 on a different directory stream.

38835 If a file is removed from or added to the directory after the most recent call to *opendir*( ) or
38836 *rewinddir*( ), whether a subsequent call to *readdir*( ) returns an entry for that file is unspecified.

38837 The *readdir*( ) function may buffer several directory entries per actual read operation; *readdir*( )
38838 shall mark for update the *st_atime* field of the directory each time the directory is actually read.

38839 After a call to *fork*( ), either the parent or child (but not both) may continue processing the
38840 XSI directory stream using *readdir*( ), *rewinddir*( ), or *seekdir*( ). If both the parent and child processes
38841 use these functions, the result is undefined.

38842 If the entry names a symbolic link, the value of the *d_ino* member is unspecified.

38843 The *readdir*( ) function need not be thread-safe. A function that is not required to be thread-safe is
38844 not required to be reentrant.

38845 The *readdir_r*( ) function shall initialize the **dirent** structure referenced by *entry* to represent the
38846 directory entry at the current position in the directory stream referred to by *dirp*, store a pointer
38847 to this structure at the location referenced by *result*, and position the directory stream at the next
38848 entry.

38849 The storage pointed to by *entry* shall be large enough for a **dirent** with an array of **char** *d_name*
38850 members containing at least {NAME_MAX}+1 elements.

38851 Upon successful return, the pointer returned at **result* shall have the same value as the argument
38852 *entry*. Upon reaching the end of the directory stream, this pointer shall have the value NULL.

38853 The *readdir_r*( ) function shall not return directory entries containing empty names.

38854 If a file is removed from or added to the directory after the most recent call to *opendir*( ) or
38855 *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*( ) and *readdir_r*( ) functions shall fail if:

[EOVERFLOW] One of the values in the structure to be returned cannot be represented correctly.

The *readdir*( ) and *readdir_r*( ) functions 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.

EXAMPLES

The following sample program searches the current directory for each of the arguments supplied on the command line.

```
#include <dirent.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>

static void lookup(const char *arg)
{
    DIR *dirp;
    struct dirent *dp;

    if ((dirp = opendir(".")) == NULL) {
        perror("couldn't open '.'");
        return;
    }

    do {
        errno = 0;
        if ((dp = readdir(dirp)) != NULL) {
            if (strcmp(dp->d_name, arg) != 0)
                continue;

            (void) printf("found %s\n", arg);
            (void) closedir(dirp);
                return;

        }
    } while (dp != NULL);

    if (errno != 0)
        perror("error reading directory");
```



```
        else
            (void) printf("failed to find %s\n", arg);
        (void) closedir(dirp);
        return;
    }

    int main(int argc, char *argv[])
    {
        int i;
        for (i = 1; i < argc; i++)
            lookup(argv[i]);
        return (0);
    }
```

APPLICATION USAGE

The *readdir*( ) function should be used in conjunction with *opendir*( ), *closedir*( ), and *rewinddir*( ) to examine the contents of the directory.

The *readdir_r*( ) function is thread-safe and shall return values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

RATIONALE

The returned value of *readdir*( ) merely *represents* a directory entry. No equivalence should be inferred.

Historical implementations of *readdir*( ) obtain multiple directory entries on a single read operation, which permits subsequent *readdir*( ) operations to operate from the buffered information. Any wording that required each successful *readdir*( ) operation to mark the directory *st_atime* field for update would disallow such historical performance-oriented implementations.

Since *readdir*( ) returns NULL when it detects an error and when the end of the directory is encountered, an application that needs to tell the difference must set *errno* to zero before the call and check it if NULL is returned. Since the function must not change *errno* in the second case and must set it to a non-zero value in the first case, a zero *errno* after a call returning NULL indicates end-of-directory; otherwise, an error.

Routines to deal with this problem more directly were proposed:

```
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.

38946 **FUTURE DIRECTIONS**

38947 None.

38948 **SEE ALSO**

38949 *closedir( )*, *dirfd( )*, *exec*, *fdopendir( )*, *fstatat( )*, *rewinddir( )*, *symlink( )*, the Base Definitions volume
38950 of IEEE Std 1003.1-200x, **<dirent.h>**, **<sys/types.h>**

38951 **CHANGE HISTORY**

38952 First released in Issue 2.

38953 **Issue 5**

38954 Large File Summit extensions are added.

38955 The *readdir_r( )* function is included for alignment with the POSIX Threads Extension.

38956 A note indicating that the *readdir( )* function need not be reentrant is added to the
38957 DESCRIPTION.

38958 **Issue 6**

38959 The *readdir_r( )* function is marked as part of the Thread-Safe Functions option.

38960 The Open Group Corrigendum U026/7 is applied, correcting the prototype for *readdir_r( )*.

38961 The Open Group Corrigendum U026/8 is applied, clarifying the wording of the successful
38962 return for the *readdir_r( )* function.

38963 The following new requirements on POSIX implementations derive from alignment with the
38964 Single UNIX Specification:

38965 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
38966 required for conforming implementations of previous POSIX specifications, it was not
38967 required for UNIX applications.

38968 • A statement is added to the DESCRIPTION indicating the disposition of certain fields in
38969 **struct dirent** when an entry refers to a symbolic link.

38970 • The [EOVERFLOW] mandatory error condition is added. This change is to support large
38971 files.

38972 • The [ENOENT] optional error condition is added.

38973 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
38974 its avoidance of possibly using a static data area.

38975 The **restrict** keyword is added to the *readdir_r( )* prototype for alignment with the
38976 ISO/IEC 9899: 1999 standard.

38977 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/50 is applied, replacing the EXAMPLES
38978 section with a new example.

38979 **Issue 7**

38980 Austin Group Interpretation 1003.1-2001 #059 is applied, updating the ERRORS section.

38981 The *readdir_r( )* function is moved from the Thread-Safe Functions option to the Base.

NAME

readlink, readlinkat — read the contents of a symbolic link relative to a directory file descriptor

SYNOPSIS

```
#include <unistd.h>

ssize_t readlink(const char *restrict path, char *restrict buf,
    size_t bufsize);
ssize_t readlinkat(int fd, const char *path, char *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.

The *readlinkat*( ) function shall be equivalent to the *readlink*( ) function except in the case where *path* specifies a relative path. In this case the symbolic link whose content is read is relative to the directory associated with the file descriptor *fd* instead of the current working directory. It is unspecified whether directory searches are permitted based on whether the file was opened with search permission or on the current permissions of the directory underlying the file descriptor.

If *readlinkat*( ) is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior shall be identical to a call to *readlink*( ).

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.

Upon successful completion, the *readlinkat*( ) function shall return 0. Otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

These functions 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 *readlinkat*( ) function shall fail if:

[EBADF] The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

These functions 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}.

The *readlinkat*( ) function may fail if:

[ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

EXAMPLES

Reading the Name of a Symbolic Link

The following example shows how to read the name of a symbolic link named **/modules/pass1**.

```
#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 1003.1-200x 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.

The purpose of the *readlinkat*( ) function is to read the content of symbolic links in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to *readlink*( ), resulting in unspecified behavior. By opening a file descriptor for the target directory and using the *readlinkat*( ) function it can be guaranteed that the symbolic link read is located relative to the desired directory.

FUTURE DIRECTIONS

None.

SEE ALSO

fstatat( ), *symlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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 **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 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.

Issue 7

The *readlinkat*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 2.

39087 **NAME**

39088 readlinkat — read the contents of a symbolic link relative to a directory file descriptor

39089 **SYNOPSIS**

39090
```
#include <unistd.h>
```

39091
```
ssize_t readlinkat(int fd, const char *path, char *buf,
```

39092
```
    size_t bufsize);
```

39093 **DESCRIPTION**

39094 Refer to *readlink( )*.

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);
...
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 1003.1-200x, **<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

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 1003.1-200x defers to the ISO C standard.

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*( ) CX shall return a null pointer and set *errno* to [ENOMEM].

ERRORS

The *realloc*( ) function shall fail if:

CX [ENOMEM] Insufficient memory is available.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

39194 **SEE ALSO**

39195 *calloc( )*, *free( )*, *malloc( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

39196 **CHANGE HISTORY**

39197 First released in Issue 1. Derived from Issue 1 of the SVID.

39198 **Issue 6**

39199 Extensions beyond the ISO C standard are marked.

39200 The following new requirements on POSIX implementations derive from alignment with the
39201 Single UNIX Specification:

39202 • In the RETURN VALUE section, if there is not enough available memory, the setting of
39203 *errno* to [ENOMEM] is added.

39204 • 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*.

If *resolved_name* is a null pointer, the behavior of *realpath*( ) is implementation-defined.

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] The *file_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 *file_name* 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 *file_name* 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.

```
#include <stdlib.h>
...
char *symlinkpath = "/tmp/symlink/file";
char actualpath [PATH_MAX+1];
char *ptr;

ptr = realpath(symlinkpath, actualpath);
```

APPLICATION USAGE

None.

RATIONALE

Since the maximum pathname length is arbitrary unless {PATH_MAX} is defined, an application generally cannot supply a *resolved_name* buffer with size {{PATH_MAX}+1}.

FUTURE DIRECTIONS

In the future, passing a null pointer to *realpath*( ) for the *resolved_name* argument may be defined to have *realpath*( ) allocate space for the generated pathname.

SEE ALSO

getcwd( ), *sysconf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/51 is applied, adding new text to the DESCRIPTION for the case when *resolved_name* is a null pointer, changing the [EINVAL] error text, adding text to the RATIONALE, and adding text to FUTURE DIRECTIONS.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/110 is applied, updating the ERRORS section to refer to the *file_name* argument, rather than a non-existent *path* argument.

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.

ERRORS

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*( ),

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

39397 received message shall be stored in the **sockaddr** structure pointed to by the *address* argument,
39398 and the length of this address shall be stored in the object pointed to by the *address_len*
39399 argument.

39400 If the actual length of the address is greater than the length of the supplied **sockaddr** structure,
39401 the stored address shall be truncated.

39402 If the *address* argument is not a null pointer and the protocol does not provide the source address
39403 of messages, the value stored in the object pointed to by *address* is unspecified.

39404 If no messages are available at the socket and O_NONBLOCK is not set on the socket's file
39405 descriptor, *recvfrom*( ) shall block until a message arrives. If no messages are available at the
39406 socket and O_NONBLOCK is set on the socket's file descriptor, *recvfrom*( ) shall fail and set *errno*
39407 to [EAGAIN] or [EWOULDBLOCK].

39408 **RETURN VALUE**

39409 Upon successful completion, *recvfrom*( ) shall return the length of the message in bytes. If no
39410 messages are available to be received and the peer has performed an orderly shutdown,
39411 *recvfrom*( ) shall return 0. Otherwise, the function shall return –1 and set *errno* to indicate the
39412 error.

39413 **ERRORS**

39414 The *recvfrom*( ) function shall fail if:

39415 [EAGAIN] or [EWOULDBLOCK]
39416 The socket's file descriptor is marked O_NONBLOCK and no data is waiting
39417 to be received; or MSG_OOB is set and no out-of-band data is available and
39418 either the socket's file descriptor is marked O_NONBLOCK or the socket does
39419 not support blocking to await out-of-band data.

39420 [EBADF] The *socket* argument is not a valid file descriptor.

39421 [ECONNRESET] A connection was forcibly closed by a peer.

39422 [EINTR] A signal interrupted *recvfrom*( ) before any data was available.

39423 [EINVAL] The MSG_OOB flag is set and no out-of-band data is available.

39424 [ENOTCONN] A receive is attempted on a connection-mode socket that is not connected.

39425 [ENOTSOCK] The *socket* argument does not refer to a socket.

39426 [EOPNOTSUPP] The specified flags are not supported for this socket type.

39427 [ETIMEDOUT] The connection timed out during connection establishment, or due to a
39428 transmission timeout on active connection.

39429 The *recvfrom*( ) function may fail if:

39430 [EIO] An I/O error occurred while reading from or writing to the file system.

39431 [ENOBUFS] Insufficient resources were available in the system to perform the operation.

39432 [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 1003.1-200x, **<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.

39490 required. The *msg_iov* and *msg_iovlen* fields are used to specify where the received data shall be
39491 stored. *msg_iov* points to an array of **iovec** structures; *msg_iovlen* shall be set to the dimension of
39492 this array. In each **iovec** structure, the *iov_base* field specifies a storage area and the *iov_len* field
39493 gives its size in bytes. Each storage area indicated by *msg_iov* is filled with received data in turn
39494 until all of the received data is stored or all of the areas have been filled.

39495 Upon successful completion, the *msg_flags* member of the message header shall be the bitwise-
39496 inclusive OR of all of the following flags that indicate conditions detected for the received
39497 message:

39498 MSG_EOR End-of-record was received (if supported by the protocol).

39499 MSG_OOB Out-of-band data was received.

39500 MSG_TRUNC Normal data was truncated.

39501 MSG_CTRUNC Control data was truncated.

39502 **RETURN VALUE**

39503 Upon successful completion, *recvmsg*( ) shall return the length of the message in bytes. If no
39504 messages are available to be received and the peer has performed an orderly shutdown,
39505 *recvmsg*( ) shall return 0. Otherwise, –1 shall be returned and *errno* set to indicate the error.

39506 **ERRORS**

39507 The *recvmsg*( ) function shall fail if:

39508 [EAGAIN] or [EWOULDBLOCK]
39509 The socket's file descriptor is marked O_NONBLOCK and no data is waiting
39510 to be received; or MSG_OOB is set and no out-of-band data is available and
39511 either the socket's file descriptor is marked O_NONBLOCK or the socket does
39512 not support blocking to await out-of-band data.

39513 [EBADF] The *socket* argument is not a valid open file descriptor.

39514 [ECONNRESET] A connection was forcibly closed by a peer.

39515 [EINTR] This function was interrupted by a signal before any data was available.

39516 [EINVAL] The sum of the *iov_len* values overflows a **ssize_t**, or the MSG_OOB flag is set
39517 and no out-of-band data is available.

39518 [EMSGSIZE] The *msg_iovlen* member of the **msghdr** structure pointed to by *message* is less
39519 than or equal to 0, or is greater than {IOV_MAX}.

39520 [ENOTCONN] A receive is attempted on a connection-mode socket that is not connected.

39521 [ENOTSOCK] The *socket* argument does not refer to a socket.

39522 [EOPNOTSUPP] The specified flags are not supported for this socket type.

39523 [ETIMEDOUT] The connection timed out during connection establishment, or due to a
39524 transmission timeout on active connection.

39525 The *recvmsg*( ) function may fail if:

39526 [EIO] An I/O error occurred while reading from or writing to the file system.

39527 [ENOBUFS] Insufficient resources were available in the system to perform the operation.

39528 [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

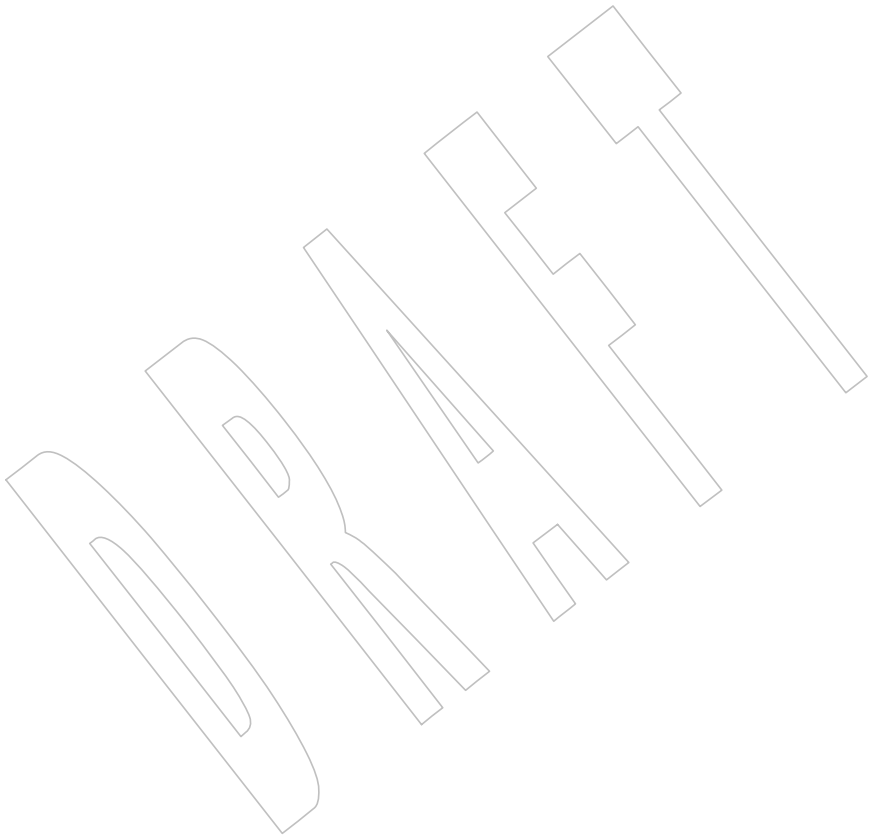
None.

SEE ALSO

poll( ), *recv*( ), *recvfrom*( ), *select*( ), *send*( ), *sendmsg*( ), *sendto*( ), *shutdown*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**

CHANGE HISTORY

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



NAME

regcomp, regerror, regexec, 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 regexec(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 Definitions volume of IEEE Std 1003.1-200x, Chapter 9, Regular Expressions.

The **regex_t** structure is defined in **<regex.h>** and contains at least the following member:

Member Type	Member Name	Description
`size_t`	`re_nsub`	Number of parenthesized subexpressions.

The **regmatch_t** structure is defined in **<regex.h>** and contains at least the following members:

Member Type	Member Name	Description
regoff_t	*rm_so*	Byte offset from start of *string* to start of substring.
regoff_t	*rm_eo*	Byte offset from start of *string* of the first character after the end of substring.

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 1003.1-200x, Chapter 9, Regular Expressions.)

REG_NOSUB Report only success/fail in *regexec*( ).

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 *regexec*( ) 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, *regexec*( ) 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:

39583 REG_NOTBOL The first character of the string pointed to by *string* is not the beginning of the
39584 line. Therefore, the circumflex character (`'^'`), when taken as a special
39585 character, shall not match the beginning of *string*.

39586 REG_NOTEOL The last character of the string pointed to by *string* is not the end of the line.
39587 Therefore, the dollar sign (`'$'`), when taken as a special character, shall not
39588 match the end of *string*.

39589 If *nmatch* is 0 or REG_NOSUB was set in the *cflags* argument to *regcomp*( ), then *regexec*( ) shall
39590 ignore the *pmatch* argument. Otherwise, the application shall ensure that the *pmatch* argument
39591 points to an array with at least *nmatch* elements, and *regexec*( ) shall fill in the elements of that
39592 array with offsets of the substrings of *string* that correspond to the parenthesized subexpressions
39593 of *pattern*: *pmatch*[*i*].*rm_so* shall be the byte offset of the beginning and *pmatch*[*i*].*rm_eo* shall be
39594 one greater than the byte offset of the end of substring *i*. (Subexpression *i* begins at the *i*th
39595 matched open parenthesis, counting from 1.) Offsets in *pmatch*[0] identify the substring that
39596 corresponds to the entire regular expression. Unused elements of *pmatch* up to *pmatch*[*nmatch*–1]
39597 shall be filled with –1. If there are more than *nmatch* subexpressions in *pattern* (*pattern* itself
39598 counts as a subexpression), then *regexec*( ) shall still do the match, but shall record only the first
39599 *nmatch* substrings.

39600 When matching a basic or extended regular expression, any given parenthesized subexpression
39601 of *pattern* might participate in the match of several different substrings of *string*, or it might not
39602 match any substring even though the pattern as a whole did match. The following rules shall be
39603 used to determine which substrings to report in *pmatch* when matching regular expressions:

39604 1. If subexpression *i* in a regular expression is not contained within another subexpression,
39605 and it participated in the match several times, then the byte offsets in *pmatch*[*i*] shall
39606 delimit the last such match.

39607 2. If subexpression *i* is not contained within another subexpression, and it did not
39608 participate in an otherwise successful match, the byte offsets in *pmatch*[*i*] shall be –1. A
39609 subexpression does not participate in the match when:

39610 `'*'` or `"\{\}"` appears immediately after the subexpression in a basic regular
39611 expression, or `'*'`, `'?'`, or `"{}"` appears immediately after the subexpression in
39612 an extended regular expression, and the subexpression did not match (matched 0
39613 times)

39614 or:

39615 `'|'` is used in an extended regular expression to select this subexpression or
39616 another, and the other subexpression matched.

39617 3. If subexpression *i* is contained within another subexpression *j*, and *i* is not contained
39618 within any other subexpression that is contained within *j*, and a match of subexpression *j*
39619 is reported in *pmatch*[*j*], then the match or non-match of subexpression *i* reported in
39620 *pmatch*[*i*] shall be as described in 1. and 2. above, but within the substring reported in
39621 *pmatch*[*j*] rather than the whole string. The offsets in *pmatch*[*i*] are still relative to the start
39622 of *string*.

39623 4. If subexpression *i* is contained in subexpression *j*, and the byte offsets in *pmatch*[*j*] are –1,
39624 then the pointers in *pmatch*[*i*] shall also be –1.

39625 5. If subexpression *i* matched a zero-length string, then both byte offsets in *pmatch*[*i*] shall be
39626 the byte offset of the character or null terminator immediately following the zero-length
39627 string.

39628 If, when *regexec*( ) is called, the locale is different from when the regular expression was
39629 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 1003.1-200x, Chapter 9, Regular Expressions).
2. A circumflex (`'^'`) in *pattern*, when used to specify expression anchoring (see the Base Definitions volume of IEEE Std 1003.1-200x, 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

```
#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 1003.1-200x, 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. This allows type-safe access to both constant and non-constant strings.

The type **regoff_t** is used for the elements of *pmatch*[ ] to ensure that the application can represent large arrays in memory (important for an application conforming to the Shell and Utilities volume of IEEE Std 1003.1-200x).

The 1992 edition of this standard required **regoff_t** to be at least as wide as **off_t**, to facilitate future extensions in which the string to be searched is taken from a file. However, these future extensions have not appeared. The requirement rules out popular implementations with 32-bit **regoff_t** and 64-bit **off_t**, so it has been withdrawn.

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 1003.1-200x, 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 1003.1-200x 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 1003.1-200x, 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 1003.1-200x 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 1003.1-200x, Section 2.13, Pattern Matching Notation, Base Definitions volume of IEEE Std 1003.1-200x, Chapter 9, Regular Expressions, **<regex.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the ISO POSIX-2 standard.

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 normative text 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.

Issue 7

SD5-XBD-ERN-60 is applied.

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

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 1003.1-200x defers to the ISO C standard.

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 $|n-x/y|=½$, the value n is chosen to be even.

The behavior of *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.

On systems that do not support the IEC 60559 Floating-Point option, if y is zero, it is implementation-defined whether a domain error occurs or zero is returned.

MX 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:

MX 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.

These functions may fail if:

Domain Error The y argument 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.

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 1003.1-200x, 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 *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.

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

Issue 7

ISO/IEC 9899: 1999 standard, Technical Corrigendum 2 #55 (SD5-XSH-ERN-82) is applied.

NAME

remove — remove a file

SYNOPSIS

```
#include <stdio.h>

int remove(const char *path);
```

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 1003.1-200x 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.

CX 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

CX Refer to *rmdir*( ) or *unlink*( ).

ERRORS

CX 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 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard and the ISO C standard.

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:

39948 • The DESCRIPTION, RETURN VALUE, and ERRORS sections are updated so that if *path* is
39949 not a directory, *remove*( ) is equivalent to *unlink*( ), and if it is a directory, it is equivalent to
39950 *rmdir*( ).

39951 **NAME**

39952 remque — remove an element from a queue

39953 **SYNOPSIS**

39954 XSI
```
#include <search.h>

void remque(void *element);
```

39956 **DESCRIPTION**

39957 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

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 1003.1-200x defers to the ISO C standard.

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. If y is zero, the value stored in the object pointed to by *quo* is unspecified.

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 REM y.

On systems that do not support the IEC 60559 Floating-Point option, if y is zero, it is implementation-defined whether a domain error occurs or zero is returned.

MX 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:

MX 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.

These functions may fail if:

Domain Error — The y argument 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.

40000 **EXAMPLES**

40001 None.

40002 **APPLICATION USAGE**

40003 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
40004 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

40005 **RATIONALE**

40006 These functions are intended for implementing argument reductions which can exploit a few
40007 low-order bits of the quotient. Note that *x* may be so large in magnitude relative to *y* that an
40008 exact representation of the quotient is not practical.

40009 **FUTURE DIRECTIONS**

40010 None.

40011 **SEE ALSO**

40012 *feclearexcept( )*, *fetestexcept( )*, *remainder( )*, the Base Definitions volume of IEEE Std 1003.1-200x,
40013 Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

40014 **CHANGE HISTORY**

40015 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

40016 **Issue 7**

40017 ISO/IEC 9899: 1999 standard, Technical Corrigendum 2 #56 (SD5-XSH-ERN-83) is applied.

40018 **NAME**

40019 rename, renameat — rename file relative to directory file descriptor

40020 **SYNOPSIS**

40021
```
#include <stdio.h>
```

40022
```
int rename(const char *old, const char *new);
```

40023 CX
```
int renameat(int oldfd, const char *old, int newfd,
    const char *new);
```

40024

40025 **DESCRIPTION**

40026 CX For *rename*( ): The functionality described on this reference page is aligned with the ISO C
40027 standard. Any conflict between the requirements described here and the ISO C standard is
40028 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

40029 The *rename*( ) function shall change the name of a file. The *old* argument points to the pathname
40030 of the file to be renamed. The *new* argument points to the new pathname of the file.

40031 CX If either the *old* or *new* argument names a symbolic link, *rename*( ) shall operate on the symbolic
40032 link itself, and shall not resolve the last component of the argument. If the *old* argument and the
40033 *new* argument resolve to the same existing file, *rename*( ) shall return successfully and perform no
40034 other action.

40035 If the *old* argument points to the pathname of a file that is not a directory, the *new* argument shall
40036 not point to the pathname of a directory. If the link named by the *new* argument exists, it shall be
40037 removed and *old* renamed to *new*. In this case, a link named *new* shall remain visible to other
40038 processes throughout the renaming operation and refer either to the file referred to by *new* or *old*
40039 before the operation began. Write access permission is required for both the directory containing
40040 *old* and the directory containing *new*.

40041 If the *old* argument points to the pathname of a directory, the *new* argument shall not point to the
40042 pathname of a file that is not a directory. If the directory named by the *new* argument exists, it
40043 shall be removed and *old* renamed to *new*. In this case, a link named *new* shall exist throughout
40044 the renaming operation and shall refer either to the directory referred to by *new* or *old* before the
40045 operation began. If *new* names an existing directory, it shall be required to be an empty directory.

40046 If either *pathname* argument refers to a path whose final component is either dot or dot-dot,
40047 *rename*( ) shall fail.

40048 If the *old* argument points to a pathname of a symbolic link, the symbolic link shall be renamed.
40049 If the *new* argument points to a pathname of a symbolic link, the symbolic link shall be removed.

40050 The *new* pathname shall not contain a path prefix that names *old*. Write access permission is
40051 required for the directory containing *old* and the directory containing *new*. If the *old* argument
40052 points to the pathname of a directory, write access permission may be required for the directory
40053 named by *old*, and, if it exists, the directory named by *new*.

40054 If the link named by the *new* argument exists and the file's link count becomes 0 when it is
40055 removed and no process has the file open, the space occupied by the file shall be freed and the
40056 file shall no longer be accessible. If one or more processes have the file open when the last link is
40057 removed, the link shall be removed before *rename*( ) returns, but the removal of the file contents
40058 shall be postponed until all references to the file are closed.

40059 Upon successful completion, *rename*( ) shall mark for update the *st_ctime* and *st_mtime* fields of
40060 the parent directory of each file.

40061 If the *rename*( ) function fails for any reason other than [EIO], any file named by *new* shall be

unaffected.

The *renameat*( ) function shall be equivalent to the *rename*( ) function except in the case where either *old* or *new* specifies a relative path. If *old* is a relative path, the file to be renamed is located relative to the directory associated with the file descriptor *oldfd* instead of the current working directory. If *new* is a relative path, the same happens only relative to the directory associated with *newfd*. It is unspecified whether directory searches are permitted based on whether the file was opened with search permission or on the current permissions of the directory underlying the file descriptor.

If *renameat*( ) is passed the special value AT_FDCWD in the *oldfd* or *newfd* parameter, the current working directory shall be used in the determination of the file for the respective *path* parameter.

RETURN VALUE

CX Upon successful completion, the *rename*( ) function shall return 0. Otherwise, it shall return –1, *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.

CX Upon successful completion, the *renameat*( ) function shall return 0. Otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

CX The *rename*( ) and *renameat*( ) functions shall fail if:

CX [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.

CX [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.

CX [EEXIST] or [ENOTEMPTY]
The link named by *new* is a directory that is not an empty directory.

CX [EINVAL] The *new* directory pathname contains a path prefix that names the *old* directory, or either *pathname* argument contains a final component that is dot or dot-dot.

CX [EIO] A physical I/O error has occurred.

CX [EISDIR] The *new* argument points to a directory and the *old* argument points to a file that is not a directory.

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

CX [EMLINK] The file named by *old* is a directory, and the link count of the parent directory of *new* would exceed {LINK_MAX}.

CX [ENAMETOOLONG]
The length of the *old* or *new* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

CX [ENOENT] The link named by *old* does not name an existing file, or either *old* or *new* points to an empty string.

CX [ENOSPC] The directory that would contain *new* cannot be extended.

CX [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.

XSI [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.

CX [EROFS] The requested operation requires writing in a directory on a read-only file system.

CX [EXDEV] The links named by *new* and *old* are on different file systems and the implementation does not support links between file systems.

CX In addition, the *renameat*( ) function shall fail if:

[EBADF] The *old* argument does not specify an absolute path and the *oldfd* argument is neither AT_FDCWD nor a valid file descriptor open for searching, or the *new* argument does not specify an absolute path and the *newfd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

CX The *rename*( ) and *renameat*( ) functions may fail if:

OB XSR [EBUSY] The file named by the *old* or *new* arguments is a named STREAM.

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

CX [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}.

CX [ETXTBSY] The file to be renamed is a pure procedure (shared text) file that is being executed.

CX The *renameat*( ) function may fail if:

[ENOTDIR] The *old* argument is not an absolute path and *oldfd* is neither AT_FDCWD nor a file descriptor associated with a directory, or the *new* argument is not an absolute path and *newfd* is neither AT_FDCWD nor a file descriptor associated with a directory.

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.

40148 **RATIONALE**

40149 This *rename*( ) function is equivalent for regular files to that defined by the ISO C standard. Its
40150 inclusion here expands that definition to include actions on directories and specifies behavior
40151 when the *new* parameter names a file that already exists. That specification requires that the
40152 action of the function be atomic.

40153 One of the reasons for introducing this function was to have a means of renaming directories
40154 while permitting implementations to prohibit the use of *link*( ) and *unlink*( ) with directories,
40155 thus constraining links to directories to those made by *mkdir*( ).

40156 The specification that if *old* and *new* refer to the same file is intended to guarantee that:

40157 `rename("x", "x");`

40158 does not remove the file.

40159 Renaming dot or dot-dot is prohibited in order to prevent cyclical file system paths.

40160 See also the descriptions of [ENOTEMPTY] and [ENAMETOOLONG] in *rmdir*( ) and [EBUSY] in
40161 *unlink*( ). For a discussion of [EXDEV], see *link*( ).

40162 The purpose of the *renameat*( ) function is to rename files in directories other than the current
40163 working directory without exposure to race conditions. Any part of the path of a file could be
40164 changed in parallel to a call to *rename*( ), resulting in unspecified behavior. By opening file
40165 descriptors for the source and target directories and using the *renameat*( ) function it can be
40166 guaranteed that that renamed file is located correctly and the resulting file is in the desired
40167 directory.

40168 **FUTURE DIRECTIONS**

40169 None.

40170 **SEE ALSO**

40171 *link*( ), *rmdir*( ), *symlink*( ), *unlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
40172 **<stdio.h>**

40173 **CHANGE HISTORY**

40174 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

40175 **Issue 5**

40176 The [EBUSY] error is added to the optional part of the ERRORS section.

40177 **Issue 6**

40178 Extensions beyond the ISO C standard are marked.

40179 The following new requirements on POSIX implementations derive from alignment with the
40180 Single UNIX Specification:

40181 - The [EIO] mandatory error condition is added.
40182 - The [ELOOP] mandatory error condition is added.
40183 - A second [ENAMETOOLONG] is added as an optional error condition.
40184 - The [ETXTBSY] optional error condition is added.

40185 The following changes were made to align with the IEEE P1003.1a draft standard:

40186 - Details are added regarding the treatment of symbolic links.
40187 - The [ELOOP] optional error condition is added.

40188 The normative text is updated to avoid use of the term "must" for application requirements.

Issue 7

Austin Group Interpretation 1003.1-2001 #076 is applied, clarifying the behavior if the final component of a path is either dot or dot-dot, and adding the associated [EINVAL] error case.

The *renameat*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 2.

40194 **NAME**

40195 renameat — rename file relative to directory file descriptor

40196 **SYNOPSIS**

40197 CX
```
#include <stdio.h>

int renameat(int oldfd, const char *old, int newfd,
    const char *new);
```

40200 **DESCRIPTION**

40201 Refer to *rename( )*.

40202 **NAME**

40203 rewind — reset the file position indicator in a stream

40204 **SYNOPSIS**

40205
```
#include <stdio.h>
```

40206
```
void rewind(FILE *stream);
```

40207 **DESCRIPTION**

40208 CX The functionality described on this reference page is aligned with the ISO C standard. Any
40209 conflict between the requirements described here and the ISO C standard is unintentional. This
40210 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

40211 The call:

40212
```
rewind(stream)
```

40213 shall be equivalent to:

40214
```
(void) fseek(stream, 0L, SEEK_SET)
```

40215 except that *rewind*( ) shall also clear the error indicator.

40216 CX Since *rewind*( ) does not return a value, an application wishing to detect errors should clear *errno*,
40217 then call *rewind*( ), and if *errno* is non-zero, assume an error has occurred.

40218 **RETURN VALUE**

40219 The *rewind*( ) function shall not return a value.

40220 **ERRORS**

40221 CX Refer to *fseek*( ) with the exception of [EINVAL] which does not apply.

40222 **EXAMPLES**

40223 None.

40224 **APPLICATION USAGE**

40225 None.

40226 **RATIONALE**

40227 None.

40228 **FUTURE DIRECTIONS**

40229 None.

40230 **SEE ALSO**

40231 *fseek*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

40232 **CHANGE HISTORY**

40233 First released in Issue 1. Derived from Issue 1 of the SVID.

40234 **Issue 6**

40235 Extensions beyond the ISO C standard are marked.

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.

XSI 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( ), *fdopendir*( ), *readdir*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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 requir

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

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 1003.1-200x defers to the ISO C standard.

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.

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 *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:

XSI

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 1003.1-200x, 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.

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

40337 **NAME**

40338 rmdir — remove a directory

40339 **SYNOPSIS**

40340 ```
#include <unistd.h>
```

40341 ```
int rmdir(const char *path);
```

40342 **DESCRIPTION**

40343 The *rmdir*( ) function shall remove a directory whose name is given by *path*. The directory shall
40344 be removed only if it is an empty directory.

40345 If the directory is the root directory or the current working directory of any process, it is
40346 unspecified whether the function succeeds, or whether it shall fail and set *errno* to [EBUSY].

40347 If *path* names a symbolic link, then *rmdir*( ) shall fail and set *errno* to [ENOTDIR].

40348 If the *path* argument refers to a path whose final component is either dot or dot-dot, *rmdir*( ) shall
40349 fail.

40350 If the directory's link count becomes 0 and no process has the directory open, the space occupied
40351 by the directory shall be freed and the directory shall no longer be accessible. If one or more
40352 processes have the directory open when the last link is removed, the dot and dot-dot entries, if
40353 present, shall be removed before *rmdir*( ) returns and no new entries may be created in the
40354 directory, but the directory shall not be removed until all references to the directory are closed.

40355 If the directory is not an empty directory, *rmdir*( ) shall fail and set *errno* to [EEXIST] or
40356 [ENOTEMPTY].

40357 Upon successful completion, the *rmdir*( ) function shall mark for update the *st_ctime* and
40358 *st_mtime* fields of the parent directory.

40359 **RETURN VALUE**

40360 Upon successful completion, the function *rmdir*( ) shall return 0. Otherwise, –1 shall be returned,
40361 and *errno* set to indicate the error. If –1 is returned, the named directory shall not be changed.

40362 **ERRORS**

40363 The *rmdir*( ) function shall fail if:

40364 [EACCES] Search permission is denied on a component of the path prefix, or write
40365 permission is denied on the parent directory of the directory to be removed.

40366 [EBUSY] The directory to be removed is currently in use by the system or some process
40367 and the implementation considers this to be an error.

40368 [EEXIST] or [ENOTEMPTY]
40369 The *path* argument names a directory that is not an empty directory, or there
40370 are hard links to the directory other than dot or a single entry in dot-dot.

40371 [EINVAL] The *path* argument contains a last component that is dot.

40372 [EIO] A physical I/O error has occurred.

40373 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
40374 argument.

40375 [ENAMETOOLONG]
40376 The length of the *path* argument exceeds {PATH_MAX} or a pathname
40377 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.

XSI [EPERM] or [EACCES]
The S_ISVTX flag is set on the parent directory of the directory to be removed and the caller is not the owner of the directory to be removed, nor is the caller the owner of the parent directory, nor does the caller have the appropriate privileges.

[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 *pathname***/dot** 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 current working directory of the process 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 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.

40423 **FUTURE DIRECTIONS**

40424 None.

40425 **SEE ALSO**

40426 Section 2.3 (on page 21), *mkdir( )*, *remove( )*, *rename( )*, *unlink( )*, the Base Definitions volume of
40427 IEEE Std 1003.1-200x, **<unistd.h>**

40428 **CHANGE HISTORY**

40429 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

40430 **Issue 6**

40431 The following new requirements on POSIX implementations derive from alignment with the
40432 Single UNIX Specification:

40433 • The DESCRIPTION is updated to indicate the results of naming a symbolic link in *path*.

40434 • The [EIO] mandatory error condition is added.

40435 • The [ELOOP] mandatory error condition is added.

40436 • A second [ENAMETOOLONG] is added as an optional error condition.

40437 The following changes were made to align with the IEEE P1003.1a draft standard:

40438 • The [ELOOP] optional error condition is added.

40439 **NAME**

40440 round, roundf, roundl — round to the nearest integer value in a floating-point format

40441 **SYNOPSIS**

40442
```
#include <math.h>
```

40443
```
double round(double x);
```
40444
```
float roundf(float x);
```
40445
```
long double roundl(long double x);
```

40446 **DESCRIPTION**

40447 CX The functionality described on this reference page is aligned with the ISO C standard. Any
40448 conflict between the requirements described here and the ISO C standard is unintentional. This
40449 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

40450 These functions shall round their argument to the nearest integer value in floating-point format,
40451 rounding halfway cases away from zero, regardless of the current rounding direction.

40452 An application wishing to check for error situations should set *errno* to zero and call
40453 *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or
40454 *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-
40455 zero, an error has occurred.

40456 **RETURN VALUE**

40457 Upon successful completion, these functions shall return the rounded integer value.

40458 MX If *x* is NaN, a NaN shall be returned.

40459 If *x* is ±0 or ±Inf, *x* shall be returned.

40460 XSI If the correct value would cause overflow, a range error shall occur and *round*( ), *roundf*( ), and
40461 *roundl*( ) shall return the value of the macro ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL
40462 (with the same sign as *x*), respectively.

40463 **ERRORS**

40464 These functions may fail if:

40465 XSI Range Error — The result overflows.

40466 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
40467 then *errno* shall be set to [ERANGE]. If the integer expression
40468 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
40469 floating-point exception shall be raised.

40470 **EXAMPLES**

40471 None.

40472 **APPLICATION USAGE**

40473 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
40474 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

40475 **RATIONALE**

40476 None.

40477 **FUTURE DIRECTIONS**

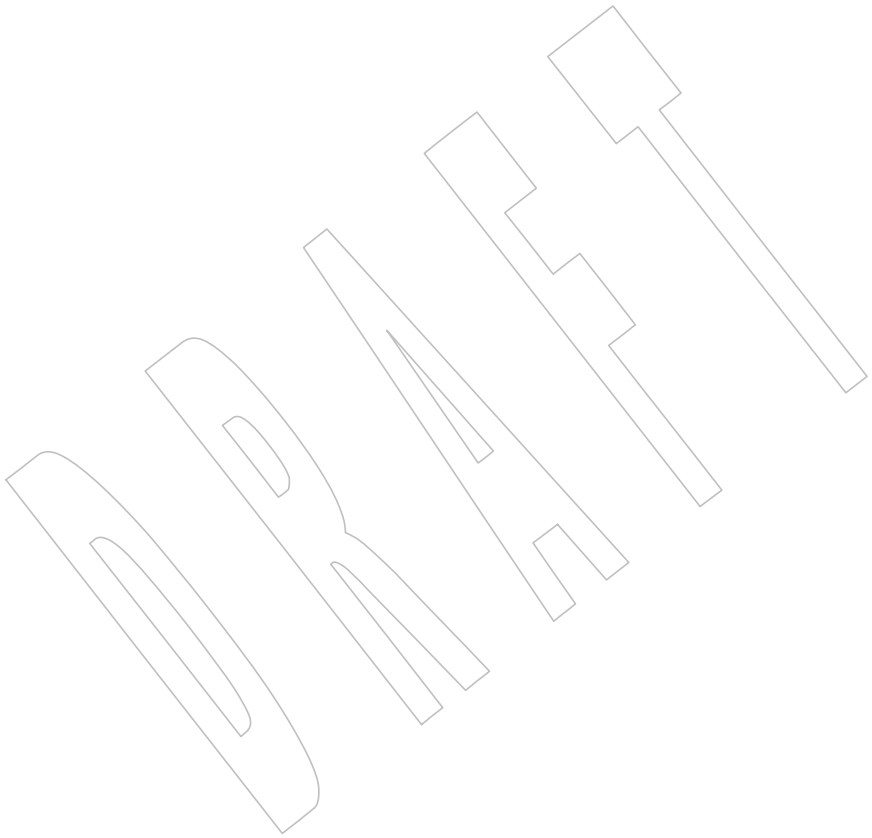
40478 None.

SEE ALSO

feclearexcept( ), *fetestexcept( )*, the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.



NAME

scalbln, scalblnf, scalblnl, scalbn, scalbnf, scalbnl — compute exponent using FLT_RADIX

SYNOPSIS

```
#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);
```

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 1003.1-200x defers to the ISO C standard.

These functions shall compute $x * \text{FLT\_RADIX}^n$ efficiently, not normally by computing FLT_RADIX^n explicitly.

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 $x * \text{FLT\_RADIX}^n$.

If the result would cause overflow, a range error shall occur and these functions shall return ±HUGE_VAL, ±HUGE_VALF, and ±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 MX either 0.0 (if supported), or an implementation-defined value shall be returned.

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

If *x* is ±0 or ±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.

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

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*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

40548 **NAME**

40549 scandir — scan a directory

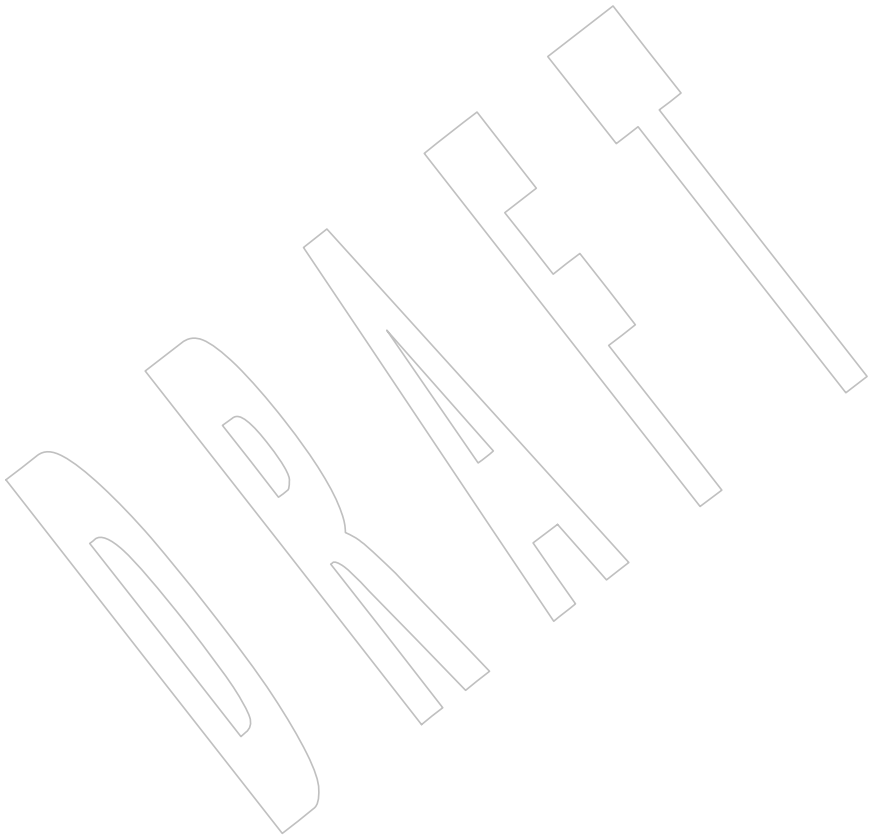
40550 **SYNOPSIS**

40551
```
#include <dirent.h>
```

40552
```
int scandir(const char *dir, struct dirent ***namelist,
    int (*sel)(const struct dirent *),
    int (*compar)(const struct dirent **, const struct dirent **));
```

40555 **DESCRIPTION**

40556 Refer to *alphasort( )*.



40557 **NAME**

40558 scanf — convert formatted input

40559 **SYNOPSIS**

40560
```
#include <stdio.h>
```

40561
```
int scanf(const char *restrict format, ... );
```

40562 **DESCRIPTION**

40563 Refer to *fscanf( )*.

40564 **NAME**

40565 sched_get_priority_max, sched_get_priority_min — get priority limits (**REALTIME**)

40566 **SYNOPSIS**

40567 PS | TPS
```
#include <sched.h>

int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```

40570 **DESCRIPTION**

40571 The *sched_get_priority_max*( ) and *sched_get_priority_min*( ) functions shall return the appropriate
40572 maximum or minimum, respectively, for the scheduling policy specified by *policy*.

40573 The value of *policy* shall be one of the scheduling policy values defined in **<sched.h>**.

40574 **RETURN VALUE**

40575 If successful, the *sched_get_priority_max*( ) and *sched_get_priority_min*( ) functions shall return the
40576 appropriate maximum or minimum values, respectively. If unsuccessful, they shall return a
40577 value of –1 and set *errno* to indicate the error.

40578 **ERRORS**

40579 The *sched_get_priority_max*( ) and *sched_get_priority_min*( ) functions shall fail if:

40580 [EINVAL] The value of the *policy* parameter does not represent a defined scheduling
40581 policy.

40582 **EXAMPLES**

40583 None.

40584 **APPLICATION USAGE**

40585 None.

40586 **RATIONALE**

40587 None.

40588 **FUTURE DIRECTIONS**

40589 None.

40590 **SEE ALSO**

40591 *sched_getparam*( ), *sched_setparam*( ), *sched_getscheduler*( ), *sched_rr_get_interval*( ),
40592 *sched_setscheduler*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sched.h>**

40593 **CHANGE HISTORY**

40594 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

40595 **Issue 6**

40596 These functions are marked as part of the Process Scheduling option.

40597 The [ENOSYS] error condition has been removed as stubs need not be provided if an
40598 implementation does not support the Process Scheduling option.

40599 The [ESRCH] error condition has been removed since these functions do not take a *pid*
40600 argument.

40601 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/52 is applied, changing the PS margin
40602 code in the SYNOPSIS to PS | TPS.

40603 **NAME**

40604 sched_getparam — get scheduling parameters (**REALTIME**)

40605 **SYNOPSIS**

40606 PS
```
#include <sched.h>
```
40607
```
int sched_getparam(pid_t pid, struct sched_param *param);
```

40608 **DESCRIPTION**

40609 The *sched_getparam*( ) function shall return the scheduling parameters of a process specified by
40610 *pid* in the **sched_param** structure pointed to by *param*.

40611 If a process specified by *pid* exists, and if the calling process has permission, the scheduling
40612 parameters for the process whose process ID is equal to *pid* shall be returned.

40613 If *pid* is zero, the scheduling parameters for the calling process shall be returned. The behavior of
40614 the *sched_getparam*( ) function is unspecified if the value of *pid* is negative.

40615 **RETURN VALUE**

40616 Upon successful completion, the *sched_getparam*( ) function shall return zero. If the call to
40617 *sched_getparam*( ) is unsuccessful, the function shall return a value of −1 and set *errno* to indicate
40618 the error.

40619 **ERRORS**

40620 The *sched_getparam*( ) function shall fail if:

40621 [EPERM] The requesting process does not have permission to obtain the scheduling
40622 parameters of the specified process.

40623 [ESRCH] No process can be found corresponding to that specified by *pid*.

40624 **EXAMPLES**

40625 None.

40626 **APPLICATION USAGE**

40627 None.

40628 **RATIONALE**

40629 None.

40630 **FUTURE DIRECTIONS**

40631 None.

40632 **SEE ALSO**

40633 *sched_getscheduler*( ), *sched_setparam*( ), *sched_setscheduler*( ), the Base Definitions volume of
40634 IEEE Std 1003.1-200x, **<sched.h>**

40635 **CHANGE HISTORY**

40636 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

40637 **Issue 6**

40638 The *sched_getparam*( ) function is marked as part of the Process Scheduling option.

40639 The [ENOSYS] error condition has been removed as stubs need not be provided if an
40640 implementation does not support the Process Scheduling option.

40641 **NAME**

40642 sched_getscheduler — get scheduling policy (**REALTIME**)

40643 **SYNOPSIS**

40644 PS
```
#include <sched.h>

int sched_getscheduler(pid_t pid);
```

40646 **DESCRIPTION**

40647 The *sched_getscheduler*( ) function shall return the scheduling policy of the process specified by
40648 *pid*. If the value of *pid* is negative, the behavior of the *sched_getscheduler*( ) function is
40649 unspecified.

40650 The values that can be returned by *sched_getscheduler*( ) are defined in the **<sched.h>** header.

40651 If a process specified by *pid* exists, and if the calling process has permission, the scheduling
40652 policy shall be returned for the process whose process ID is equal to *pid*.

40653 If *pid* is zero, the scheduling policy shall be returned for the calling process.

40654 **RETURN VALUE**

40655 Upon successful completion, the *sched_getscheduler*( ) function shall return the scheduling policy
40656 of the specified process. If unsuccessful, the function shall return –1 and set *errno* to indicate the
40657 error.

40658 **ERRORS**

40659 The *sched_getscheduler*( ) function shall fail if:

40660 [EPERM] The requesting process does not have permission to determine the scheduling
40661 policy of the specified process.

40662 [ESRCH] No process can be found corresponding to that specified by *pid*.

40663 **EXAMPLES**

40664 None.

40665 **APPLICATION USAGE**

40666 None.

40667 **RATIONALE**

40668 None.

40669 **FUTURE DIRECTIONS**

40670 None.

40671 **SEE ALSO**

40672 *sched_getparam*( ), *sched_setparam*( ), *sched_setscheduler*( ), the Base Definitions volume of
40673 IEEE Std 1003.1-200x, **<sched.h>**

40674 **CHANGE HISTORY**

40675 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

40676 **Issue 6**

40677 The *sched_getscheduler*( ) function is marked as part of the Process Scheduling option.

40678 The [ENOSYS] error condition has been removed as stubs need not be provided if an
40679 implementation does not support the Process Scheduling option.

NAME

sched_rr_get_interval — get execution time limits (**REALTIME**)

SYNOPSIS

PS | TPS

```
#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 1003.1-200x, **<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.

IEEE Std 1003.1-2001/Cor 1-2002, XSH/TC1/D6/53 is applied, changing the PS margin code in the SYNOPSIS to PS | TPS.

NAME

sched_setparam — set scheduling parameters (**REALTIME**)

SYNOPSIS

PS
```
#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.

See Scheduling Policies for a description on how this function affects the scheduling of the threads within the target process.

SS If the current scheduling policy for the target process is not SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC, the result is implementation-defined; this case includes the SCHED_OTHER policy.

SS 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.

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.

SEE ALSO

Scheduling Policies (on page 44), *sched_getparam( )*, *sched_getscheduler( )*, *sched_setscheduler( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

Issue 7

Austin Group Interpretation 1003.1-2001 #061 is applied, updating the DESCRIPTION.

NAME

sched_setscheduler — set scheduling policy and parameters (**REALTIME**)

SYNOPSIS

PS
```
#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 the scheduling policy of the process, or the scheduling policy of another process, 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.

See Scheduling Policies for a description on how this function affects the scheduling of the threads within the target process.

SS If the current scheduling policy for the target process is not SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC, the result is implementation-defined; this case includes the SCHED_OTHER policy.

SS 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.

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.

40824 **RETURN VALUE**

40825 Upon successful completion, the function shall return the former scheduling policy of the
40826 specified process. If the *sched_setscheduler*( ) function fails to complete successfully, the policy
40827 and scheduling parameters shall remain unchanged, and the function shall return a value of –1
40828 and set *errno* to indicate the error.

40829 **ERRORS**

40830 The *sched_setscheduler*( ) function shall fail if:

40831 [EINVAL] The value of the *policy* parameter is invalid, or one or more of the parameters
40832 contained in *param* is outside the valid range for the specified scheduling
40833 policy.

40834 [EPERM] The requesting process does not have permission to set either or both of the
40835 scheduling parameters or the scheduling policy of the specified process.

40836 [ESRCH] No process can be found corresponding to that specified by *pid*.

40837 **EXAMPLES**

40838 None.

40839 **APPLICATION USAGE**

40840 None.

40841 **RATIONALE**

40842 None.

40843 **FUTURE DIRECTIONS**

40844 None.

40845 **SEE ALSO**

40846 Scheduling Policies (on page 44), *sched_getparam*( ), *sched_getscheduler*( ), *sched_setparam*( ), the
40847 Base Definitions volume of IEEE Std 1003.1-200x, **<sched.h>**

40848 **CHANGE HISTORY**

40849 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

40850 **Issue 6**

40851 The *sched_setscheduler*( ) function is marked as part of the Process Scheduling option.

40852 The [ENOSYS] error condition has been removed as stubs need not be provided if an
40853 implementation does not support the Process Scheduling option.

40854 The following new requirements on POSIX implementations derive from alignment with the
40855 Single UNIX Specification:

40856 • In the DESCRIPTION, the effect of this function on a thread's scheduling parameters is
40857 added.

40858 • Sections describing two-level scheduling and atomicity of the function are added.

40859 The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

40860 **Issue 7**

40861 Austin Group Interpretation 1003.1-2001 #061 is applied, updating the DESCRIPTION.

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

40897 **NAME**

40898 seed48 — seed a uniformly distributed pseudo-random non-negative long integer generator

40899 **SYNOPSIS**

40900 XSI
```
#include <stdlib.h>

unsigned short *seed48(unsigned short seed16v[3]);
```

40902 **DESCRIPTION**

40903 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 option.

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

fdopendir( ), *readdir*( ), *telldir*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<dirent.h>**, **<stdio.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 2.

Issue 6

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

40947 **NAME**

40948 select — synchronous I/O multiplexing

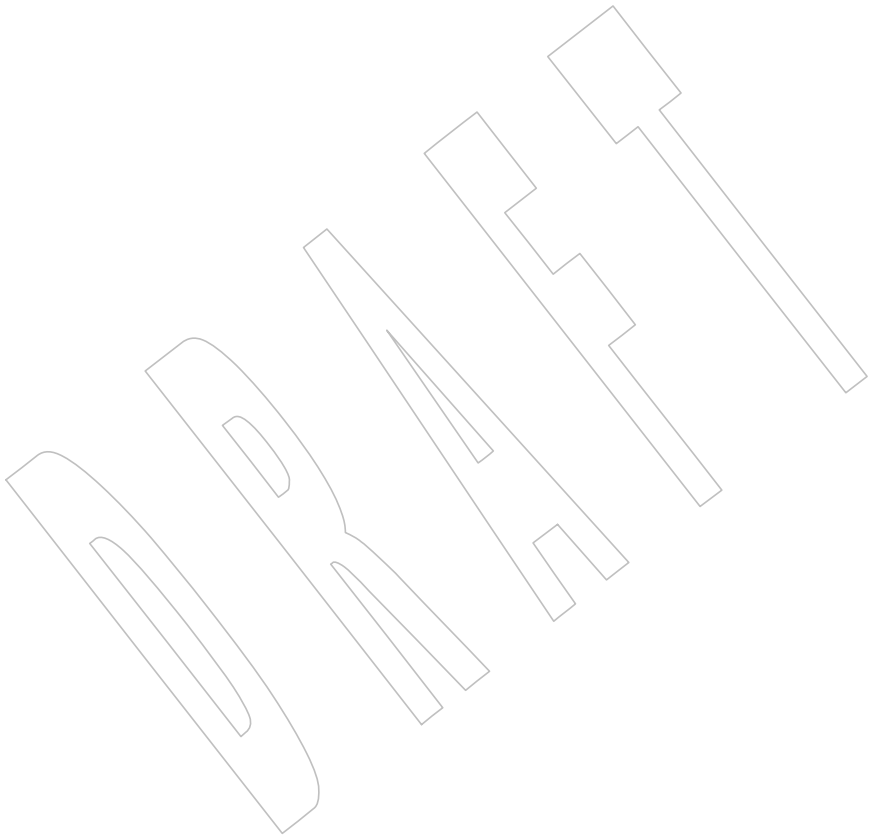
40949 **SYNOPSIS**

40950
```
#include <sys/select.h>
```

40951
```
int select(int nfds, fd_set *restrict readfds,
    fd_set *restrict writefds, fd_set *restrict errorfds,
    struct timeval *restrict timeout);
```

40954 **DESCRIPTION**

40955 Refer to *pselect( )*.



40956 **NAME**

40957 sem_close — close a named semaphore

40958 **SYNOPSIS**

40959 ```
#include <semaphore.h>
```

40960 ```
int sem_close(sem_t *sem);
```

40961 **DESCRIPTION**

40962 The *sem_close*( ) function shall indicate that the calling process is finished using the named
40963 semaphore indicated by *sem*. The effects of calling *sem_close*( ) for an unnamed semaphore (one
40964 created by *sem_init*( )) are undefined. The *sem_close*( ) function shall deallocate (that is, make
40965 available for reuse by a subsequent *sem_open*( ) by this process) any system resources allocated
40966 by the system for use by this process for this semaphore. The effect of subsequent use of the
40967 semaphore indicated by *sem* by this process is undefined. If the semaphore has not been
40968 removed with a successful call to *sem_unlink*( ), then *sem_close*( ) has no effect on the state of the
40969 semaphore. If the *sem_unlink*( ) function has been successfully invoked for *name* after the most
40970 recent call to *sem_open*( ) with O_CREAT for this semaphore, then when all processes that have
40971 opened the semaphore close it, the semaphore is no longer accessible.

40972 **RETURN VALUE**

40973 Upon successful completion, a value of zero shall be returned. Otherwise, a value of –1 shall be
40974 returned and *errno* set to indicate the error.

40975 **ERRORS**

40976 The *sem_close*( ) function may fail if:

40977 [EINVAL] The *sem* argument is not a valid semaphore descriptor.

40978 **EXAMPLES**

40979 None.

40980 **APPLICATION USAGE**

40981 The *sem_close*( ) function is part of the Semaphores option and need not be available on all
40982 implementations.

40983 **RATIONALE**

40984 None.

40985 **FUTURE DIRECTIONS**

40986 None.

40987 **SEE ALSO**

40988 *semctl*( ), *semget*( ), *semop*( ), *sem_init*( ), *sem_open*( ), *sem_unlink*( ), the Base Definitions volume of
40989 IEEE Std 1003.1-200x, **<semaphore.h>**

40990 **CHANGE HISTORY**

40991 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

40992 **Issue 6**

40993 The *sem_close*( ) function is marked as part of the Semaphores option.

40994 The [ENOSYS] error condition has been removed as stubs need not be provided if an
40995 implementation does not support the Semaphores option.

40996 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/113 is applied, updating the ERRORS
40997 section so that the [EINVAL] error becomes optional.

Issue 7

The *sem_close*( ) function is moved from the Semaphores option to the Base.

41000 **NAME**

41001 sem_destroy — destroy an unnamed semaphore

41002 **SYNOPSIS**

41003
```
#include <semaphore.h>
```

41004
```
int sem_destroy(sem_t *sem);
```

41005 **DESCRIPTION**

41006 The *sem_destroy*( ) function shall destroy the unnamed semaphore indicated by *sem*. Only a
41007 semaphore that was created using *sem_init*( ) may be destroyed using *sem_destroy*( ); the effect of
41008 calling *sem_destroy*( ) with a named semaphore is undefined. The effect of subsequent use of the
41009 semaphore *sem* is undefined until *sem* is reinitialized by another call to *sem_init*( ).

41010 It is safe to destroy an initialized semaphore upon which no threads are currently blocked. The
41011 effect of destroying a semaphore upon which other threads are currently blocked is undefined.

41012 **RETURN VALUE**

41013 Upon successful completion, a value of zero shall be returned. Otherwise, a value of –1 shall be
41014 returned and *errno* set to indicate the error.

41015 **ERRORS**

41016 The *sem_destroy*( ) function may fail if:

41017 [EINVAL] The *sem* argument is not a valid semaphore.

41018 [EBUSY] There are currently processes blocked on the semaphore.

41019 **EXAMPLES**

41020 None.

41021 **APPLICATION USAGE**

41022 The *sem_destroy*( ) function is part of the Semaphores option and need not be available on all
41023 implementations.

41024 **RATIONALE**

41025 None.

41026 **FUTURE DIRECTIONS**

41027 None.

41028 **SEE ALSO**

41029 *semctl*( ), *semget*( ), *semop*( ), *sem_init*( ), *sem_open*( ), the Base Definitions volume of
41030 IEEE Std 1003.1-200x, **<semaphore.h>**

41031 **CHANGE HISTORY**

41032 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

41033 **Issue 6**

41034 The *sem_destroy*( ) function is marked as part of the Semaphores option.

41035 The [ENOSYS] error condition has been removed as stubs need not be provided if an
41036 implementation does not support the Semaphores option.

41037 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/114 is applied, updating the ERRORS
41038 section so that the [EINVAL] error becomes optional.

Issue 7

The *dem_destroy*( ) function is moved from the Semaphores option to the Base.

NAME

sem_getvalue — get the value of a semaphore

SYNOPSIS

```
#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 object to which *sval* points shall either be set to zero or to 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 may 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 1003.1-200x, **<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.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/54 is applied.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/115 is applied, updating the ERRORS section so that the [EINVAL] error becomes optional.

Issue 7

The *sem_getvalue*( ) function is moved from the Semaphores option to the Base.

NAME

sem_init — initialize an unnamed semaphore

SYNOPSIS

```
#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_timedwait*( ), *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_timedwait*( ), *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_timedwait*( ), *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_timedwait*( ), *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 1003.1-200x 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.

41129 **SEE ALSO**

41130 *sem_destroy( )*, *sem_post( )*, *sem_timedwait( )*, *sem_trywait( )*, *sem_wait( )*, the Base Definitions
41131 volume of IEEE Std 1003.1-200x, **<semaphore.h>**

41132 **CHANGE HISTORY**

41133 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

41134 **Issue 6**

41135 The *sem_init( )* function is marked as part of the Semaphores option.

41136 The [ENOSYS] error condition has been removed as stubs need not be provided if an
41137 implementation does not support the Semaphores option.

41138 The *sem_timedwait( )* function is added to the SEE ALSO section for alignment with IEEE Std
41139 1003.1d-1999.

41140 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/116 is applied, updating the
41141 DESCRIPTION to add the *sem_timedwait( )* function for alignment with IEEE Std 1003.1d-1999.

41142 **Issue 7**

41143 The *sem_init( )* function is moved from the Semaphores option to the Base.

41144 **NAME**

41145 sem_open — initialize and open a named semaphore

41146 **SYNOPSIS**

41147 ```
#include <semaphore.h>
```

41148 ```
sem_t *sem_open(const char *name, int oflag, ...);
```

41149 **DESCRIPTION**

41150 The *sem_open*( ) function shall establish a connection between a named semaphore and a process.
41151 Following a call to *sem_open*( ) with semaphore name *name*, the process may reference the
41152 semaphore associated with *name* using the address returned from the call. This semaphore may
41153 be used in subsequent calls to *sem_wait*( ), *sem_timedwait*( ), *sem_trywait*( ), *sem_post*( ), and
41154 *sem_close*( ). The semaphore remains usable by this process until the semaphore is closed by a
41155 successful call to *sem_close*( ), *_exit*( ), or one of the *exec* functions.

41156 The *oflag* argument controls whether the semaphore is created or merely accessed by the call to
41157 *sem_open*( ). The following flag bits may be set in *oflag*:

41158 O_CREAT This flag is used to create a semaphore if it does not already exist. If O_CREAT is
41159 set and the semaphore already exists, then O_CREAT has no effect, except as noted
41160 under O_EXCL. Otherwise, *sem_open*( ) creates a named semaphore. The O_CREAT
41161 flag requires a third and a fourth argument: *mode*, which is of type **mode_t**, and
41162 *value*, which is of type **unsigned**. The semaphore is created with an initial value of
41163 *value*. Valid initial values for semaphores are less than or equal to
41164 {SEM_VALUE_MAX}.

41165 The user ID of the semaphore is set to the effective user ID of the process; the
41166 group ID of the semaphore is set to a system default group ID or to the effective
41167 group ID of the process. The permission bits of the semaphore are set to the value
41168 of the *mode* argument except those set in the file mode creation mask of the process.
41169 When bits in *mode* other than the file permission bits are specified, the effect is
41170 unspecified.

41171 After the semaphore named *name* has been created by *sem_open*( ) with the
41172 O_CREAT flag, other processes can connect to the semaphore by calling
41173 *sem_open*( ) with the same value of *name*.

41174 O_EXCL If O_EXCL and O_CREAT are set, *sem_open*( ) fails if the semaphore *name* exists.
41175 The check for the existence of the semaphore and the creation of the semaphore if it
41176 does not exist are atomic with respect to other processes executing *sem_open*( ) with
41177 O_EXCL and O_CREAT set. If O_EXCL is set and O_CREAT is not set, the effect is
41178 undefined.

41179 If flags other than O_CREAT and O_EXCL are specified in the *oflag* parameter, the
41180 effect is unspecified.

41181 The *name* argument points to a string naming a semaphore object. It is unspecified whether the
41182 name appears in the file system and is visible to functions that take pathnames as arguments.
41183 The *name* argument conforms to the construction rules for a pathname, except that the
41184 interpretation of slash characters other than the leading slash character in *name* is
41185 implementation-defined, and that the length limits for the *name* argument are implementation-
41186 defined and need not be the same as the path name limits {PATH_MAX} and {NAME_MAX}. If
41187 *name* begins with the slash character, then processes calling *sem_open*( ) with the same value of
41188 *name* shall refer to the same semaphore object, as long as that name has not been removed. If
41189 *name* does not begin with the slash character, the effect 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 have been no calls to *sem_unlink*( ) for this semaphore, and at least one previous successful *sem_open*( ) call for this semaphore has not been matched with a *sem_close*( ) call.

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}.

[EMFILE] Too many semaphore descriptors or file descriptors are currently in use by this process.

[ENFILE] Too many semaphores are currently open in the system.

[ENOENT] O_CREAT is not set and the named semaphore does not exist.

[ENOMEM] There is insufficient memory for the creation of the new named semaphore.

[ENOSPC] There is insufficient space on a storage device for the creation of the new named semaphore.

If any of the following conditions occur, the *sem_open*( ) function may return SEM_FAILED and set *errno* to the corresponding value:

[ENAMETOOLONG]
The length of the *name* argument exceeds {_POSIX_PATH_MAX} on systems that do not support the XSI option or exceeds {_XOPEN_PATH_MAX} on XSI systems, or has a pathname component that is longer than {_POSIX_NAME_MAX} on systems that do not support the XSI option or longer than {_XOPEN_NAME_MAX} on XSI systems.

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.

41234 **FUTURE DIRECTIONS**

41235 None.

41236 **SEE ALSO**

41237 *semctl*( ), *semget*( ), *semop*( ), *sem_close*( ), *sem_post*( ), *sem_timedwait*( ), *sem_trywait*( ), *sem_unlink*( ),
41238 *sem_wait*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<semaphore.h>**

41239 **CHANGE HISTORY**

41240 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

41241 **Issue 6**

41242 The *sem_open*( ) function is marked as part of the Semaphores option.

41243 The [ENOSYS] error condition has been removed as stubs need not be provided if an
41244 implementation does not support the Semaphores option.

41245 The *sem_timedwait*( ) function is added to the SEE ALSO section for alignment with IEEE Std
41246 1003.1d-1999.

41247 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/117 is applied, updating the
41248 DESCRIPTION to add the *sem_timedwait*( ) function for alignment with IEEE Std 1003.1d-1999.

41249 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/118 is applied, updating the
41250 DESCRIPTION to describe the conditions to return the same semaphore address on a call to
41251 *sem_open*( ). The words "and at least one previous successful *sem_open*( ) call for this semaphore
41252 has not been matched with a *sem_close*( ) call" are added.

41253 **Issue 7**

41254 Austin Group Interpretation 1003.1-2001 #066 is applied, updating the [ENOSPC] error case and
41255 adding the [ENOMEM] error case.

41256 Austin Group Interpretation 1003.1-2001 #077 is applied, clarifying the *name* argument and
41257 adding [ENAMETOOLONG] as a "may fail" error.

41258 The *sem_open*( ) function is moved from the Semaphores option to the Base.

41259 **NAME**

41260 sem_post — unlock a semaphore

41261 **SYNOPSIS**

41262
```
#include <semaphore.h>
```

41263
```
int sem_post(sem_t *sem);
```

41264 **DESCRIPTION**

41265 The *sem_post*( ) function shall unlock the semaphore referenced by *sem* by performing a
41266 semaphore unlock operation on that semaphore.

41267 If the semaphore value resulting from this operation is positive, then no threads were blocked
41268 waiting for the semaphore to become unlocked; the semaphore value is simply incremented.

41269 If the value of the semaphore resulting from this operation is zero, then one of the threads
41270 blocked waiting for the semaphore shall be allowed to return successfully from its call to
41271 PS *sem_wait*( ). If the Process Scheduling option is supported, the thread to be unblocked shall be
41272 chosen in a manner appropriate to the scheduling policies and parameters in effect for the
41273 blocked threads. In the case of the schedulers SCHED_FIFO and SCHED_RR, the highest
41274 priority waiting thread shall be unblocked, and if there is more than one highest priority thread
41275 blocked waiting for the semaphore, then the highest priority thread that has been waiting the
41276 longest shall be unblocked. If the Process Scheduling option is not defined, the choice of a thread
41277 to unblock is unspecified.

41278 SS If the Process Sporadic Server option is supported, and the scheduling policy is
41279 SCHED_SPORADIC, the semantics are as per SCHED_FIFO above.

41280 The *sem_post*( ) function shall be reentrant with respect to signals and may be invoked from a
41281 signal-catching function.

41282 **RETURN VALUE**

41283 If successful, the *sem_post*( ) function shall return zero; otherwise, the function shall return –1
41284 and set *errno* to indicate the error.

41285 **ERRORS**

41286 The *sem_post*( ) function may fail if:

41287 [EINVAL] The *sem* argument does not refer to a valid semaphore.

41288 **EXAMPLES**

41289 None.

41290 **APPLICATION USAGE**

41291 The *sem_post*( ) function is part of the Semaphores option and need not be available on all
41292 implementations.

41293 **RATIONALE**

41294 None.

41295 **FUTURE DIRECTIONS**

41296 None.

41297 **SEE ALSO**

41298 *semctl*( ), *semget*( ), *semop*( ), *sem_timedwait*( ), *sem_trywait*( ), *sem_wait*( ), the Base Definitions
41299 volume of IEEE Std 1003.1-200x, Section 4.10, Memory Synchronization, **<semaphore.h>**

41300 **CHANGE HISTORY**

41301 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

41302 **Issue 6**

41303 The *sem_post*( ) function is marked as part of the Semaphores option.

41304 41305 The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.

41306 41307 The *sem_timedwait*( ) function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

41308 41309 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.

41310 41311 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/119 is applied, updating the ERRORS section so that the [EINVAL] error becomes optional.

41312 **Issue 7**

41313 The *sem_post*( ) function is moved from the Semaphores option to the Base.

41314 **NAME**

41315 sem_timedwait — lock a semaphore

41316 **SYNOPSIS**

41317 ```
#include <semaphore.h>
```
41318 ```
#include <time.h>
```

41319 ```
int sem_timedwait(sem_t *restrict sem,
```
41320 ```
    const struct timespec *restrict abs_timeout);
```

41321 **DESCRIPTION**

41322 The *sem_timedwait*( ) function shall lock the semaphore referenced by *sem* as in the *sem_wait*( )
41323 function. However, if the semaphore cannot be locked without waiting for another process or
41324 thread to unlock the semaphore by performing a *sem_post*( ) function, this wait shall be
41325 terminated when the specified timeout expires.

41326 The timeout shall expire when the absolute time specified by *abs_timeout* passes, as measured by
41327 the clock on which timeouts are based (that is, when the value of that clock equals or exceeds
41328 *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time
41329 of the call.

41330 The timeout shall be based on the CLOCK_REALTIME clock. The resolution of the timeout shall
41331 be the resolution of the clock on which it is based. The **timespec** data type is defined as a
41332 structure in the **<time.h>** header.

41333 Under no circumstance shall the function fail with a timeout if the semaphore can be locked
41334 immediately. The validity of the *abs_timeout* need not be checked if the semaphore can be locked
41335 immediately.

41336 **RETURN VALUE**

41337 The *sem_timedwait*( ) function shall return zero if the calling process successfully performed the
41338 semaphore lock operation on the semaphore designated by *sem*. If the call was unsuccessful, the
41339 state of the semaphore shall be unchanged, and the function shall return a value of −1 and set
41340 *errno* to indicate the error.

41341 **ERRORS**

41342 The *sem_timedwait*( ) function shall fail if:

41343 [EINVAL] The process or thread would have blocked, and the *abs_timeout* parameter
41344 specified a nanoseconds field value less than zero or greater than or equal to
41345 1 000 million.

41346 [ETIMEDOUT] The semaphore could not be locked before the specified timeout expired.

41347 The *sem_timedwait*( ) function may fail if:

41348 [EDEADLK] A deadlock condition was detected.

41349 [EINTR] A signal interrupted this function.

41350 [EINVAL] The *sem* argument does not refer to a valid semaphore.

EXAMPLES

None.

APPLICATION USAGE

Applications using these functions may be subject to priority inversion, as discussed in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

sem_post( ), *sem_trywait( )*, *sem_wait( )*, *semctl( )*, *semget( )*, *semop( )*, *time( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<semaphore.h>**, **<time.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/120 is applied, updating the ERRORS section so that the [EINVAL] error becomes optional.

Issue 7

The *sem_timedwait( )* function is moved from the Semaphores option to the Base.

Functionality relating to the Timers option is moved to the Base.

41370 **NAME**

41371 sem_trywait, sem_wait — lock a semaphore

41372 **SYNOPSIS**

41373
```
#include <semaphore.h>
```

41374
```
int sem_trywait(sem_t *sem);
```
41375
```
int sem_wait(sem_t *sem);
```

41376 **DESCRIPTION**

41377 The *sem_trywait*( ) function shall lock the semaphore referenced by *sem* only if the semaphore is
41378 currently not locked; that is, if the semaphore value is currently positive. Otherwise, it shall not
41379 lock the semaphore.

41380 The *sem_wait*( ) function shall lock the semaphore referenced by *sem* by performing a semaphore
41381 lock operation on that semaphore. If the semaphore value is currently zero, then the calling
41382 thread shall not return from the call to *sem_wait*( ) until it either locks the semaphore or the call is
41383 interrupted by a signal.

41384 Upon successful return, the state of the semaphore shall be locked and shall remain locked until
41385 the *sem_post*( ) function is executed and returns successfully.

41386 The *sem_wait*( ) function is interruptible by the delivery of a signal.

41387 **RETURN VALUE**

41388 The *sem_trywait*( ) and *sem_wait*( ) functions shall return zero if the calling process successfully
41389 performed the semaphore lock operation on the semaphore designated by *sem*. If the call was
41390 unsuccessful, the state of the semaphore shall be unchanged, and the function shall return a
41391 value of −1 and set *errno* to indicate the error.

41392 **ERRORS**

41393 The *sem_trywait*( ) function shall fail if:

41394 [EAGAIN] The semaphore was already locked, so it cannot be immediately locked by the
41395 *sem_trywait*( ) operation.

41396 The *sem_trywait*( ) and *sem_wait*( ) functions may fail if:

41397 [EDEADLK] A deadlock condition was detected.

41398 [EINTR] A signal interrupted this function.

41399 [EINVAL] The *sem* argument does not refer to a valid semaphore.

41400 **EXAMPLES**

41401 None.

41402 **APPLICATION USAGE**

41403 Applications using these functions may be subject to priority inversion, as discussed in the Base
41404 Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

41405 The *sem_trywait*( ) and *sem_wait*( ) functions are part of the Semaphores option and need not be
41406 provided on all implementations.

41407 **RATIONALE**

41408 None.

41409 **FUTURE DIRECTIONS**

41410 None.

41411 **SEE ALSO**

41412 *semctl*( ), *semget*( ), *semop*( ), *sem_post*( ), *sem_timedwait*( ), the Base Definitions volume of
41413 IEEE Std 1003.1-200x, Section 4.10, Memory Synchronization, **<semaphore.h>**

41414 **CHANGE HISTORY**

41415 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

41416 **Issue 6**

41417 The *sem_trywait*( ) and *sem_wait*( ) functions are marked as part of the Semaphores option.

41418 The [ENOSYS] error condition has been removed as stubs need not be provided if an
41419 implementation does not support the Semaphores option.

41420 The *sem_timedwait*( ) function is added to the SEE ALSO section for alignment with IEEE Std
41421 1003.1d-1999.

41422 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/121 is applied, updating the ERRORS
41423 section so that the [EINVAL] error becomes optional.

41424 **Issue 7**

41425 SD5-XSH-ERN-54 is applied, removing the *sem_wait*( ) function from the ''shall fail'' error cases.

41426 The *sem_trywait*( ) and *sem_wait*( ) functions are moved from the Semaphores option to the Base.

NAME

sem_unlink — remove a named semaphore

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.

[ENOENT] The named semaphore does not exist.

The *sem_unlink*( ) function may fail if:

[ENAMETOOLONG]
The length of the *name* argument exceeds {_POSIX_PATH_MAX} on systems
XSI that do not support the XSI option or exceeds {_XOPEN_PATH_MAX} on XSI systems, or has a pathname component that is longer than
XSI {_POSIX_NAME_MAX} on systems that do not support the XSI option or longer than {_XOPEN_NAME_MAX} on XSI systems. A call to *sem_unlink*( ) with a *name* argument that contains the same semaphore name as was previously used in a successful *sem_open*( ) call shall not give an [ENAMETOOLONG] error.

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 1003.1-200x, **<semaphore.h>**

CHANGE HISTORY

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

Issue 6

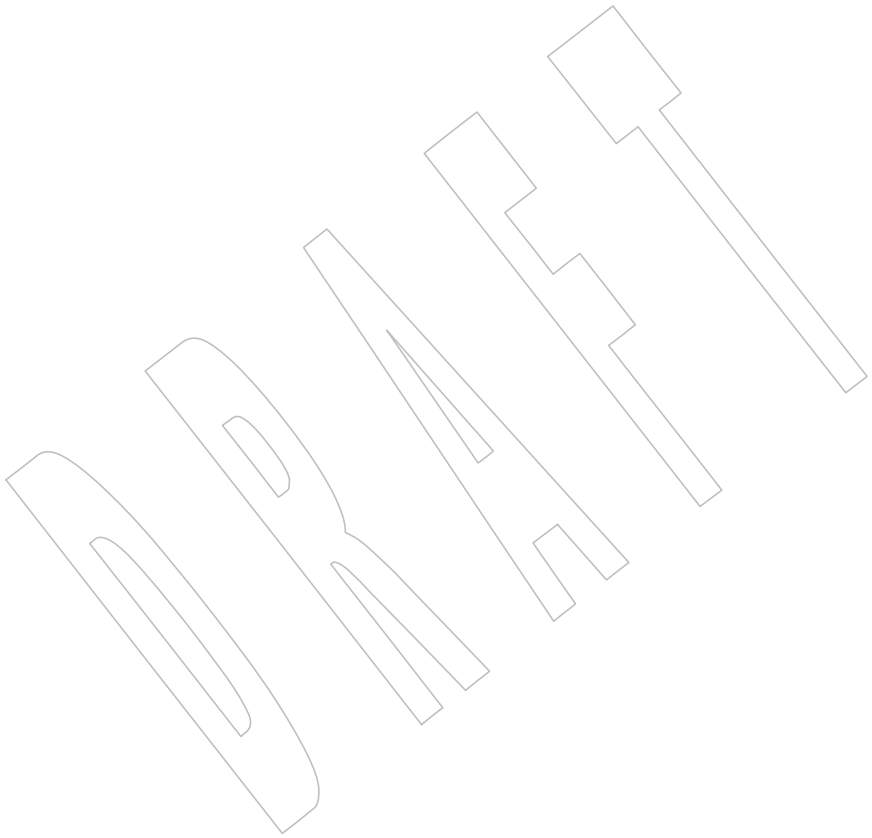
The *sem_unlink*( ) function is marked as part of the Semaphores option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.

Issue 7

Austin Group Interpretation 1003.1-2001 #077 is applied, changing [ENAMETOOLONG] from a "shall fail" to a "may fail" error.

The *sem_unlink*( ) function is moved from the Semaphores option to the Base.



41481 **NAME**

41482 sem_wait — lock a semaphore

41483 **SYNOPSIS**

41484 `#include <semaphore.h>`

41485 `int sem_wait(sem_t *sem);`

41486 **DESCRIPTION**

41487 Refer to *sem_trywait( )*.

NAME

semctl — XSI semaphore control operations

SYNOPSIS

XSI
```
#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 1003.1-200x, Section 4.15, Semaphore). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 40).

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

41531 **<sys/sem.h>**. Requires read permission.

41532 IPC_SET Set the value of the following members of the **semid_ds** data structure
41533 associated with *semid* to the corresponding value found in the structure
41534 pointed to by *arg.buf*, where *arg* is the fourth argument to *semctl*( ):

41535 `sem_perm.uid`
41536 `sem_perm.gid`
41537 `sem_perm.mode`

41538 The mode bits specified in Section 2.7.1 are copied into the corresponding bits
41539 of the *sem_perm.mode* associated with *semid*. The stored values of any other
41540 bits are unspecified.

41541 This command can only be executed by a process that has an effective user ID
41542 equal to either that of a process with appropriate privileges or to the value of
41543 *sem_perm.cuid* or *sem_perm.uid* in the **semid_ds** data structure associated with
41544 *semid*.

41545 IPC_RMID Remove the semaphore identifier specified by *semid* from the system and
41546 destroy the set of semaphores and **semid_ds** data structure associated with it.
41547 This command can only be executed by a process that has an effective user ID
41548 equal to either that of a process with appropriate privileges or to the value of
41549 *sem_perm.cuid* or *sem_perm.uid* in the **semid_ds** data structure associated with
41550 *semid*.

41551 **RETURN VALUE**

41552 If successful, the value returned by *semctl*( ) depends on *cmd* as follows:

41553 GETVAL The value of *semval*.

41554 GETPID The value of *sempid*.

41555 GETNCNT The value of *semncnt*.

41556 GETZCNT The value of *semzcnt*.

41557 All others 0.

41558 Otherwise, *semctl*( ) shall return −1 and set *errno* to indicate the error.

41559 **ERRORS**

41560 The *semctl*( ) function shall fail if:

41561 [EACCES] Operation permission is denied to the calling process; see Section 2.7 (on page
41562 39).

41563 [EINVAL] The value of *semid* is not a valid semaphore identifier, or the value of *semnum*
41564 is less than 0 or greater than or equal to *sem_nsems*, or the value of *cmd* is not a
41565 valid command.

41566 [EPERM] The argument *cmd* is equal to IPC_RMID or IPC_SET and the effective user ID
41567 of the calling process is not equal to that of a process with appropriate
41568 privileges and it is not equal to the value of *sem_perm.cuid* or *sem_perm.uid* in
41569 the data structure associated with *semid*.

41570 [ERANGE] The argument *cmd* is equal to SETVAL or SETALL and the value to which
41571 *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 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 40), *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 1003.1-200x, **<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 1003.1-200x, Section 4.15, Semaphore). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 40).

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 need 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).

[EEXIST] 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.

[ENOENT] A semaphore identifier does not exist for the argument *key* and (*semflg* &IPC_CREAT) is equal to 0.

[ENOSPC] A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphores system-wide would be exceeded.

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.

```
#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;
        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 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 40), *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 1003.1-200x, **<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.

Issue 6

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/122 is applied, updating the DESCRIPTION from "each semaphore in the set shall not be initialized" to "each semaphore in the set need not be initialized".

NAME

semop — XSI semaphore operations

SYNOPSIS

XSI
```
#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 1003.1-200x, Section 4.15, Semaphore). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 40).

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:

Member Type	Member Name	Description
short	*sem_num*	Semaphore number.
short	*sem_op*	Semaphore operation.
short	*sem_flg*	Operation flags.

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 *semadj* value of the calling process 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 *semadj* value of the calling process 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 returned.

— 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 *semadj* value of the calling process 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.

[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.

```
#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.

```
#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;
        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 can be easily modified to use the alternative interfaces.

41907 **RATIONALE**

41908 None.

41909 **FUTURE DIRECTIONS**

41910 None.

41911 **SEE ALSO**

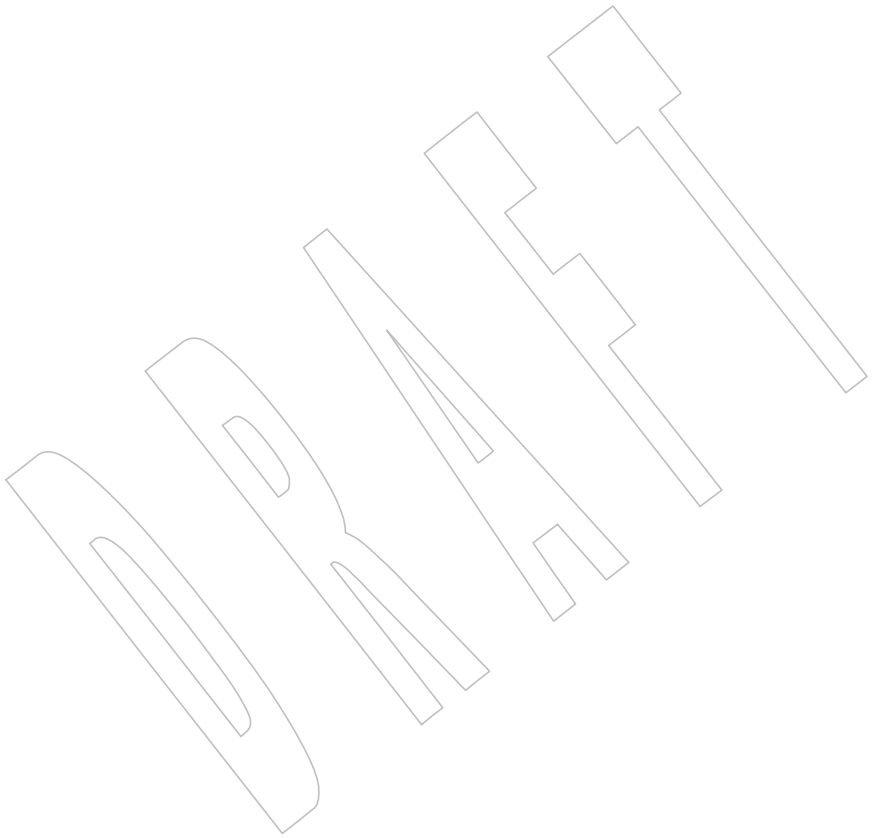
41912 Section 2.7 (on page 39), Section 2.8 (on page 40), *exec*, *exit( )*, *fork( )*, *semctl( )*, *semget( )*,
41913 *sem_close( )*, *sem_destroy( )*, *sem_getvalue( )*, *sem_init( )*, *sem_open( )*, *sem_post( )*, *sem_unlink( )*,
41914 *sem_wait( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/ipc.h>**, **<sys/sem.h>**,
41915 **<sys/types.h>**

41916 **CHANGE HISTORY**

41917 First released in Issue 2. Derived from Issue 2 of the SVID.

41918 **Issue 5**

41919 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
41920 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. If the socket is a connectionless-mode socket, the message shall be sent to the pre-specified peer address.

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.
- MSG_NOSIGNAL — Requests not to send the SIGPIPE signal if an attempt to send is made on a stream-oriented socket that is no longer connected. The [EPIPE] error shall still be returned.

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.

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

[ECONNRESET] A connection was forcibly closed by a peer.

[EDESTADDRREQ] The socket is not connection-mode and no peer address is set.

[EINTR] A signal interrupted *send*( ) before any data was transmitted.

[EMSGSIZE] The message is too large to be sent all at once, as the socket requires.

[ENOTCONN] The socket 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.

The *send*( ) function may fail if:

[EACCES] The calling process does not have the appropriate privileges.

[EIO] An I/O error occurred while reading from or writing to the file system.

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

[ENETUNREACH] No route to the network is present.

[ENOBUFS] Insufficient resources were available in the system to perform the operation.

EXAMPLES

None.

APPLICATION USAGE

The *send*( ) function is equivalent to *sendto*( ) with a null pointer *dest_len* argument, and to *write*( ) if no flags are used.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

connect( ), *getsockopt*( ), *poll*( ), *recv*( ), *recvfrom*( ), *recvmsg*( ), *select*( ), *sendmsg*( ), *sendto*( ), *setsockopt*( ), *shutdown*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**

CHANGE HISTORY

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

Issue 7

Austin Group Interpretation 1003.1-2001 #035 is applied, updating the DESCRIPTION to clarify the behavior when the socket is a connectionless-mode socket.

The MSG_NOSIGNAL flag is added from The Open Group Technical Standard, 2006, Extended API Set Part 2.

NAME

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 a connectionless-mode socket, the message shall be sent to the address specified by **msghdr** if no pre-specified peer address has been set. If a peer address has been pre-specified, either the message shall be sent to the address specified in **msghdr** (overriding the pre-specified peer address), or the function shall return –1 and set *errno* to [EISCONN]. 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.

MSG_NOSIGNAL Requests not to send the SIGPIPE signal if an attempt to send is made on a stream-oriented socket that is no longer connected. The [EPIPE] error shall still be returned.

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.

[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 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.

[ENETUNREACH]
No route to the network is present.

[ENOBUFS] 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 *sendmsg*( ) 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

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 1003.1-200x, **<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.

Issue 7

Austin Group Interpretation 1003.1-2001 #073 is applied, updating the DESCRIPTION.

The MSG_NOSIGNAL flag is added from The Open Group Technical Standard, 2006, Extended API Set Part 2.

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 a connectionless-mode socket, the message shall be sent to the address specified by *dest_addr* if no pre-specified peer address has been set. If a peer address has been pre-specified, either the message shall be sent to the address specified by *dest_addr* (overriding the pre-specified peer address), or the function shall return –1 and set *errno* to [EISCONN]. 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.
- MSG_NOSIGNAL Requests not to send the SIGPIPE signal if an attempt to send is made on a stream-oriented socket that is no longer connected. The [EPIPE] error shall still be returned.

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.

ERRORS

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.

[EINTR] A signal interrupted *sendto*( ) before any data was transmitted.

[EMSGSIZE] The message is too large to be sent all at once, as the socket requires.

[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 *sendto*( ) 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.

[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.

The *sendto*( ) 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).

[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.

[EISCONN] A destination address was specified and the socket is already connected.

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

[ENETUNREACH] No route to the network is present.

[ENOBUFS] 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 1003.1-200x, **<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.

Issue 7

Austin Group Interpretations 1003.1-2001 #035 and #073 are applied, updating the [EISCONN] error and the DESCRIPTION.

The MSG_NOSIGNAL flag is added from The Open Group Technical Standard, 2006, Extended API Set Part 2.

NAME

setbuf — assign buffering to a stream

SYNOPSIS

```
#include <stdio.h>

void setbuf(FILE *restrict stream, char *restrict buf);
```

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 1003.1-200x defers to the ISO C standard.

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 1003.1-200x, **<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

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 1003.1-200x, **<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

CX
```
#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 thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

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

See *exec*, for restrictions on changing the environment in multi-threaded applications.

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 1003.1-200x 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 part of the XSI option.

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

exec, *getenv*( ), *unsetenv*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the IEEE P1003.1a draft standard.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/55 is applied, adding references to *exec* in the APPLICATION USAGE and SEE ALSO sections.

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 user ID or the saved set-user-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*( ), *setgid*( ), *setregid*( ), *setreuid*( ), *setuid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the IEEE P1003.1a draft standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/123 is applied, making an editorial correction to the [EPERM] error in the ERRORS section.

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*( ), *setregid*( ), *setreuid*( ), *setuid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

- 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

XSI

```
#include <grp.h>

void setgrent(void);
```

DESCRIPTION

Refer to *endgrent( )*.

42458 **NAME**

42459 sethostent — network host database functions

42460 **SYNOPSIS**

42461
```
#include <netdb.h>
```

42462
```
void sethostent(int stayopen);
```

42463 **DESCRIPTION**

42464 Refer to *endhostent( )*.

NAME

setitimer — set the value of an interval timer

SYNOPSIS

OB XSI

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

int setitimer(int which, const struct itimerval *restrict value,
    struct itimerval *restrict ovalue);
```

DESCRIPTION

Refer to *getitimer( )*.

NAME

setjmp — set jump point for a non-local goto

SYNOPSIS

```
#include <setjmp.h>

int setjmp(jmp_buf env);
```

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 1003.1-200x 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.

RETURN VALUE

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.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

In general, *sigsetjmp*( ) is more useful in dealing with errors and interrupts encountered in a low-level subroutine of a program.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

longjmp( ), *sigsetjmp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<setjmp.h>**

CHANGE HISTORY

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

Issue 6

The normative text is updated to avoid use of the term "must" for application requirements.

NAME

setkey — set encoding key (**CRYPT**)

SYNOPSIS

XSI

```
#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 thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

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 1003.1-200x, **<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

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 1003.1-200x defers to the ISO C standard.

The *setlocale*( ) function selects the appropriate piece of the locale of the process, as specified by the *category* and *locale* arguments, and may be used to change or query the entire locale of the process or portions thereof. The value *LC_ALL* for *category* names the entire locale of the process; other values for *category* name only a part of the locale of the process:

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.

CX *LC_MESSAGES* Affects what strings are expected by commands and utilities as affirmative or negative responses.

XSI It also affects what strings are given by commands and utilities as affirmative or negative responses, and the content of messages.

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*:

CX "POSIX" Specifies the minimal environment for C-language translation called the POSIX locale. If *setlocale*( ) is not invoked, the POSIX locale is the default at entry to *main*( ).

"C" Equivalent to `"POSIX"`.

CX "" Specifies an implementation-defined native environment. The determination of the name of the new locale for the specified category depends on the value of the associated environment variables, *LC_** and *LANG*; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale and the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables.

A null pointer Used to direct *setlocale*( ) to query the current internationalized environment and return the name of the locale.

CX Setting all of the categories of the locale of the process is similar to successively setting each individual category of the locale of the process, except that all error checking is done before any actions are performed. To set all the categories of the locale of the process, *setlocale*( ) is invoked as:

```
setlocale(LC_ALL, "");
```

In this case, *setlocale*( ) shall first verify that the values of all the environment variables it needs according to the precedence rules (described in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables) indicate supported locales. If the value of any of these environment variable searches yields a locale that is not supported (and non-null), *setlocale*( ) shall return a null pointer and the locale of the process shall not be changed. If all environment variables name supported locales, *setlocale*( ) shall proceed as if it had been called for each category, using the appropriate value from the associated environment variable or from the implementation-defined default if there is no such value.

The locale state is common to all threads within a process.

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 locale of the process is not changed.

A null pointer for *locale* causes *setlocale*( ) to return a pointer to the string associated with the *category* for the current locale of the process. The locale of the process 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 locale of the process. 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 locale of the process such that regular expressions and string operations can be applied to text recorded in a different language:

```
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:

```
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:

```
char *setlocale(int category, const char *locale);
```

where *category* is the name of one of following categories, namely:

LC_COLLATE
LC_CTYPE
LC_MESSAGES
LC_MONETARY
LC_NUMERIC
LC_TIME

In addition, a special value called *LC_ALL* directs *setlocale*( ) to set all categories.

There are two primary uses of *setlocale*( ):

1. Querying the international environment to find out what it is set to
2. Setting the international environment, or *locale*, to a specific value

The behavior of *setlocale*( ) in these two areas is described below. Since it is difficult to describe the behavior in words, examples are used to illustrate the behavior of specific uses.

To query the international environment, *setlocale*( ) is invoked with a specific category and the NULL pointer as the locale. The NULL pointer is a special directive to *setlocale*( ) that tells it to query rather than set the international environment. The following syntax is used to query the name of the international environment:

```
setlocale({LC_ALL, LC_COLLATE, LC_CTYPE, LC_MESSAGES, LC_MONETARY, \
    LC_NUMERIC, LC_TIME},(char *) NULL);
```

The *setlocale*( ) function shall return the string corresponding to the current international environment. This value may be used by a subsequent call to *setlocale*( ) to reset the international environment to this value. However, it should be noted that the return value from *setlocale*( ) may be a pointer to a static area within the function and is not guaranteed to remain unchanged (that is, it may be modified by a subsequent call to *setlocale*( )). Therefore, if the purpose of calling *setlocale*( ) is to save the value of the current international environment so it can be changed and reset later, the return value should be copied to an array of **char** in the calling program.

There are three ways to set the international environment with *setlocale*( ):

setlocale(*category*, *string*)

This usage sets a specific *category* in the international environment to a specific value corresponding to the value of the *string*. A specific example is provided below:

```
setlocale(LC_ALL, "fr_FR.ISO-8859-1");
```

In this example, all categories of the international environment are set to the locale corresponding to the string `"fr_FR.ISO-8859-1"`, or to the French language as spoken in France using the ISO/IEC 8859-1: 1998 standard codeset.

If the string does not correspond to a valid locale, *setlocale*( ) shall return a NULL pointer and the international environment is not changed. Otherwise, *setlocale*( ) shall return the name of the locale just set.

setlocale(*category*, "C")
The ISO C standard states that one locale must exist on all conforming implementations. The name of the locale is C and corresponds to a minimal international environment needed to support the C programming language.

setlocale(*category*, "")
This sets a specific category to an implementation-defined default. This corresponds to the value of the environment variables.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *isalnum*( ), *isalpha*( ), *isblank*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ), *iswalnum*( ), *iswalpha*( ), *iswblank*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ), *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *isxdigit*( ), *localeconv*( ), *mblen*( ), *mbstowcs*( ), *mbtowc*( ), *nl_langinfo*( ), *printf*( ), *scanf*( ), *setlocale*( ), *strcoll*( ), *strerror*( ), *strfmon*( ), *strsignal*( ), *strtod*( ), *strxfrm*( ), *tolower*( ), *toupper*( ), *towlower*( ), *towupper*( ), *uselocale*( ), *wcscoll*( ), *wcstod*( ), *wcstombs*( ), *wcsxfrm*( ), *wctomb*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<langinfo.h>**, **<locale.h>**

CHANGE HISTORY

First released in Issue 3.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

Extensions beyond the ISO C standard are marked.

The normative text is updated to avoid use of the term "must" for application requirements.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/124 is applied, updating the DESCRIPTION to clarify the behavior of:

```
setlocale(LC_ALL, "");
```

Issue 7

Functionality relating to the Threads option is moved to the Base.

NAME

setlogmask — set the log priority mask

SYNOPSIS

XSI

```
#include <syslog.h>

int setlogmask(int maskpri);
```

DESCRIPTION

Refer to *closelog( )*.

42720 **NAME**

42721 setnetent — network database function

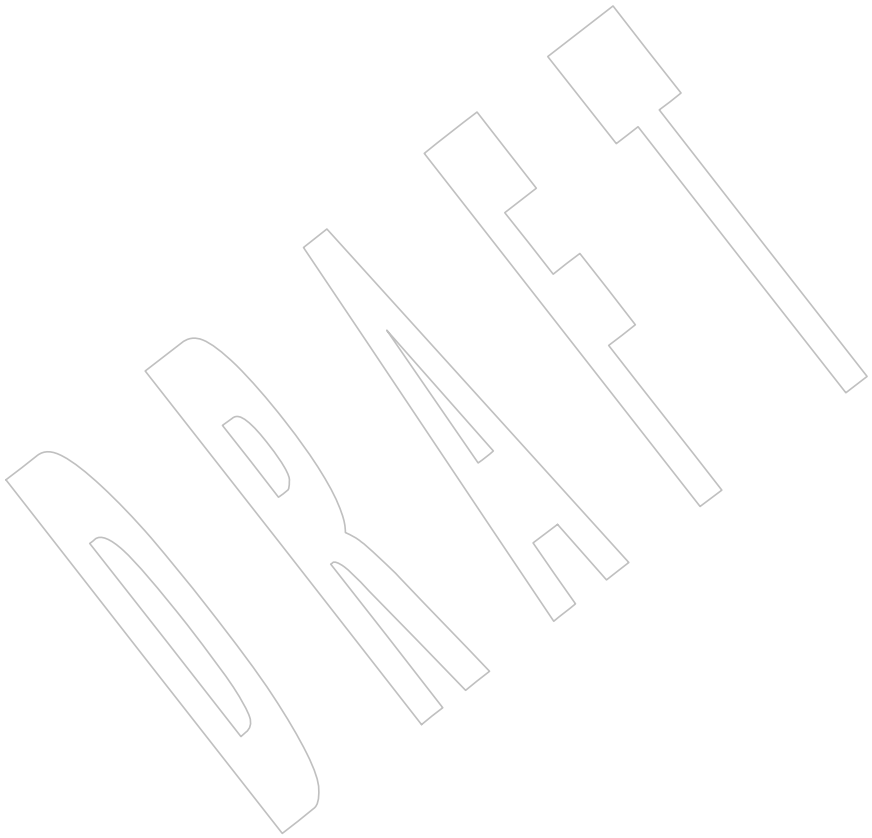
42722 **SYNOPSIS**

42723
```
#include <netdb.h>
```

42724
```
void setnetent(int stayopen);
```

42725 **DESCRIPTION**

42726 Refer to *endnetent*( ).



NAME

setpgid — set process group ID for job control

SYNOPSIS

```
#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 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

NAME

setpgrp — set the process group ID

SYNOPSIS

OB XSI
```
#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

It is unspecified whether this function behaves as *setpgid*(0,0) or *setsid*( ) unless the process is already a session leader. Therefore, applications are encouraged to use *setpgid*( ) or *setsid*( ) as appropriate.

RATIONALE

None.

FUTURE DIRECTIONS

The *setpgrp*( ) function may be removed in a future version.

SEE ALSO

exec, *fork*( ), *getpid*( ), *getsid*( ), *kill*( ), *setpgid*( ), *setsid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 7

The *setpgrp*( ) function is marked obsolescent.

42848 **NAME**

42849 setpriority — set the nice value

42850 **SYNOPSIS**

42851 XSI
```
#include <sys/resource.h>

int setpriority(int which, id_t who, int nice);
```

42853 **DESCRIPTION**

42854 Refer to *getpriority( )*.

42855 **NAME**

42856 setprotoent — network protocol database functions

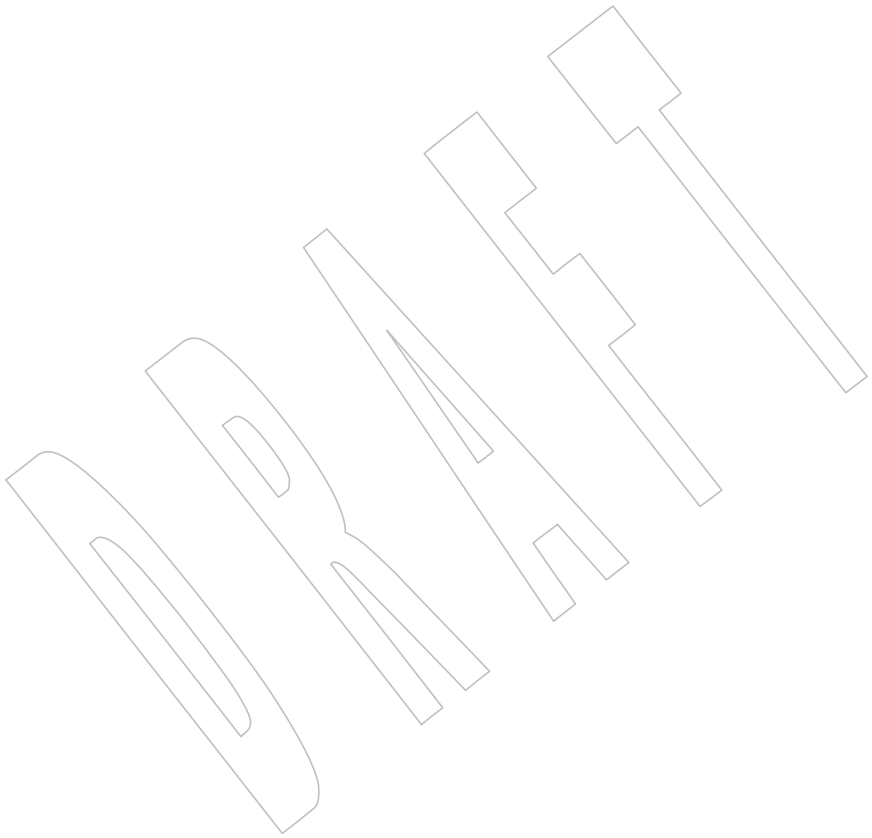
42857 **SYNOPSIS**

42858
```
#include <netdb.h>
```

42859
```
void setprotoent(int stayopen);
```

42860 **DESCRIPTION**

42861 Refer to *endprotoent( )*.



42862 **NAME**

42863 setpwent — user database function

42864 **SYNOPSIS**

42865 XSI
```
#include <pwd.h>

void setpwent(void);
```

42867 **DESCRIPTION**

42868 Refer to *endpwent( )*.

NAME

setregid — set real and effective group IDs

SYNOPSIS

XSI

```
#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, *getegid*( ), *geteuid*( ), *getgid*( ), *getuid*( ), *setegid*( ), *seteuid*( ), *setgid*( ), *setreuid*( ), *setuid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

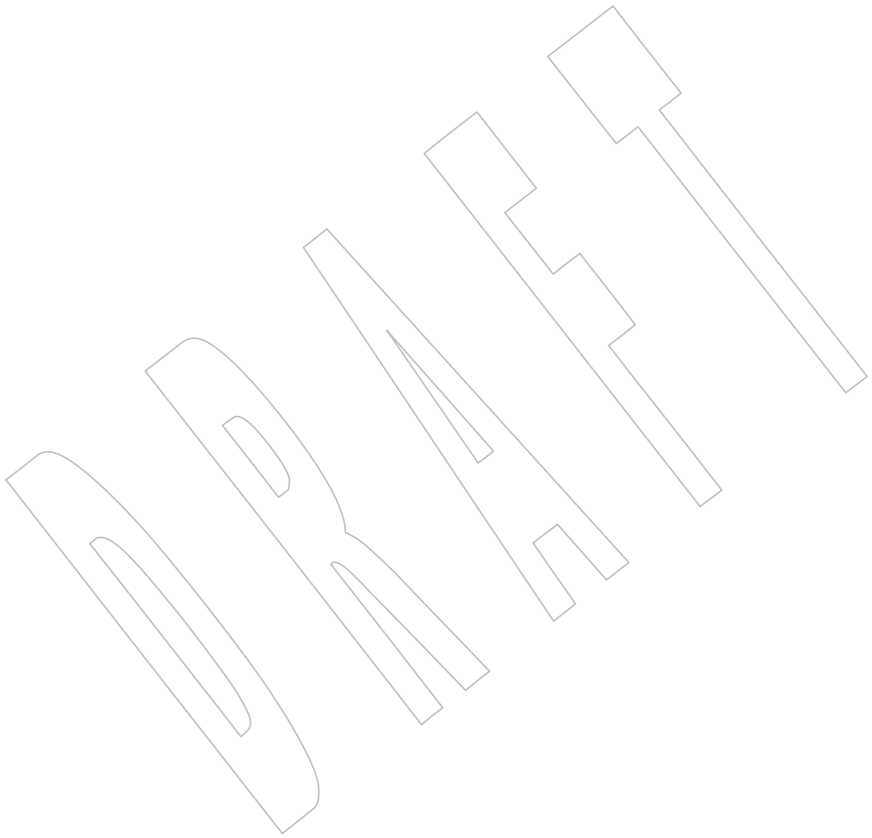
42907 **CHANGE HISTORY**

42908 First released in Issue 4, Version 2.

42909 **Issue 5**

42910 Moved from X/OPEN UNIX extension to BASE.

42911 The DESCRIPTION is updated to indicate that the saved set-group-ID can be set by any of the
42912 *exec* family of functions, not just *execve*( ).



NAME

setreuid — set real and effective user IDs

SYNOPSIS

XSI
```
#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.

42952 **SEE ALSO**

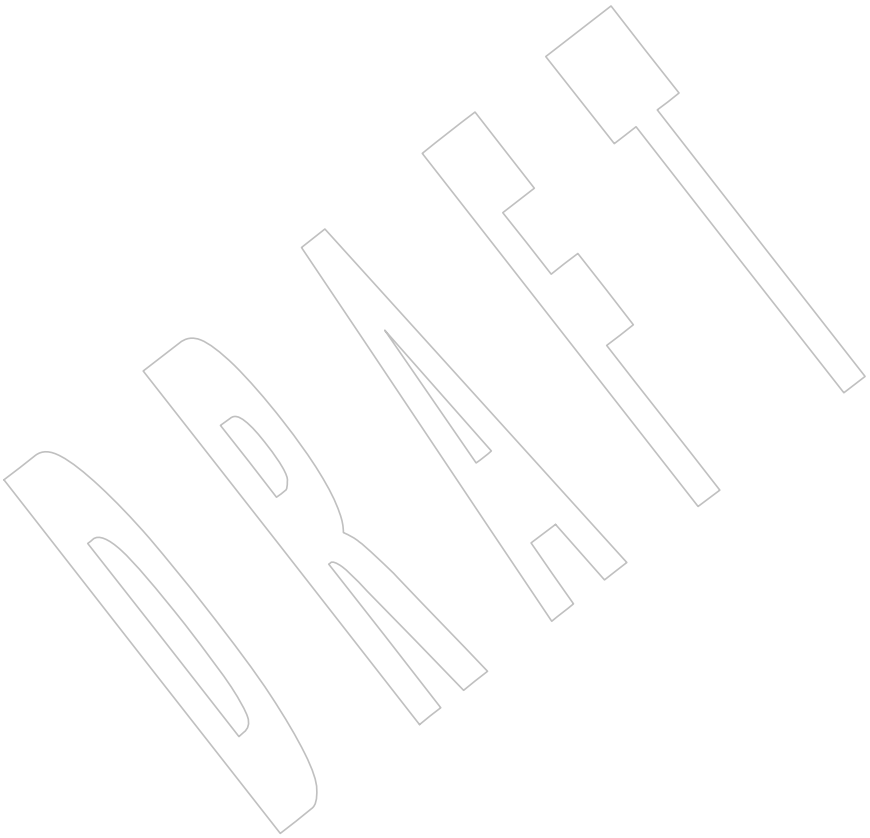
42953 *getegid( )*, *geteuid( )*, *getgid( )*, *getuid( )*, *setegid( )*, *seteuid( )*, *setgid( )*, *setregid( )*, *setuid( )*, the Base
42954 Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

42955 **CHANGE HISTORY**

42956 First released in Issue 4, Version 2.

42957 **Issue 5**

42958 Moved from X/OPEN UNIX extension to BASE.



42959 **NAME**

42960 setrlimit — control maximum resource consumption

42961 **SYNOPSIS**

42962 XSI
```
#include <sys/resource.h>

int setrlimit(int resource, const struct rlimit *rlp);
```

42963

42964 **DESCRIPTION**

42965 Refer to *getrlimit( )*.

42966 **NAME**

42967 setservent — network services database functions

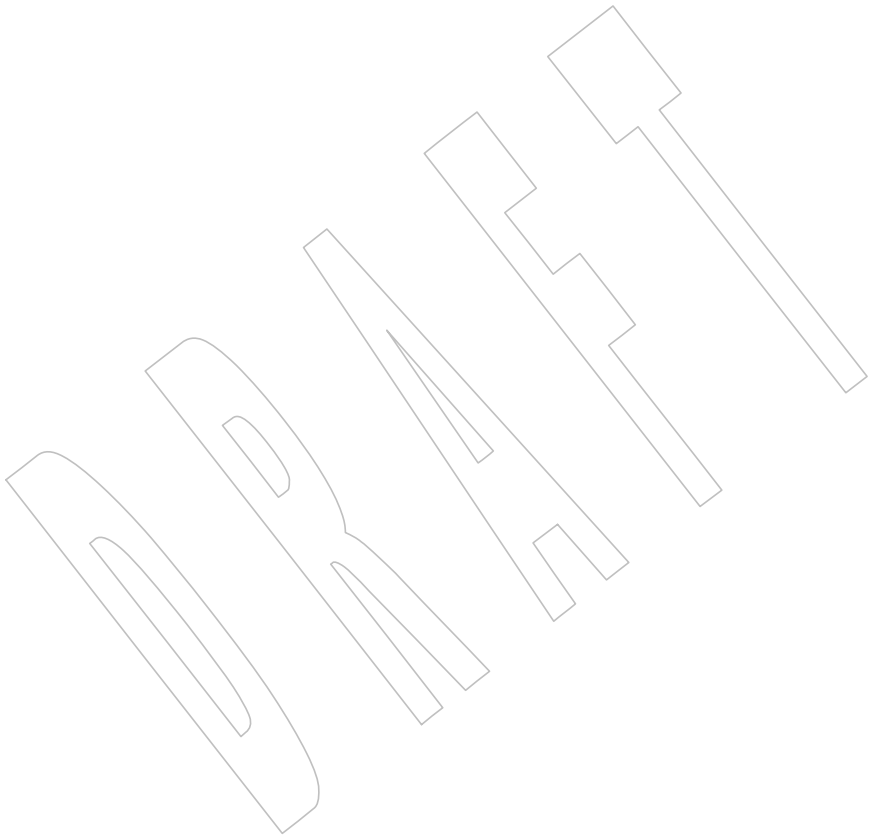
42968 **SYNOPSIS**

42969
```
#include <netdb.h>
```

42970
```
void setservent(int stayopen);
```

42971 **DESCRIPTION**

42972 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 1003.1-200x 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 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

43015 **CHANGE HISTORY**

43016 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

43017 **Issue 6**

43018 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

43019 The following new requirements on POSIX implementations derive from alignment with the
43020 Single UNIX Specification:

43021 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
43022 required for conforming implementations of previous POSIX specifications, it was not
43023 required for UNIX applications.

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 calling thread 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 calling thread 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.

43067 SO_SNDBUF Sets send buffer size. This option takes an **int** value.

43068 SO_RCVBUF Sets receive buffer size. This option takes an **int** value.

43069 SO_DONTROUTE Requests that outgoing messages bypass the standard routing facilities.
43070 The destination shall be on a directly-connected network, and messages
43071 are directed to the appropriate network interface according to the
43072 destination address. The effect, if any, of this option depends on what
43073 protocol is in use. This option takes an **int** value. This is a Boolean option.

43074 SO_RCVLOWAT Sets the minimum number of bytes to process for socket input operations.
43075 The default value for SO_RCVLOWAT is 1. If SO_RCVLOWAT is set to a
43076 larger value, blocking receive calls normally wait until they have received
43077 the smaller of the low water mark value or the requested amount. (They
43078 may return less than the low water mark if an error occurs, a signal is
43079 caught, or the type of data next in the receive queue is different from that
43080 returned; for example, out-of-band data.) This option takes an **int** value.
43081 Note that not all implementations allow this option to be set.

43082 SO_RCVTIMEO Sets the timeout value that specifies the maximum amount of time an
43083 input function waits until it completes. It accepts a **timeval** structure with
43084 the number of seconds and microseconds specifying the limit on how
43085 long to wait for an input operation to complete. If a receive operation has
43086 blocked for this much time without receiving additional data, it shall
43087 return with a partial count or *errno* set to [EAGAIN] or
43088 [EWOULDBLOCK] if no data is received. The default for this option is
43089 zero, which indicates that a receive operation shall not time out. This
43090 option takes a **timeval** structure. Note that not all implementations allow
43091 this option to be set.

43092 SO_SNDLOWAT Sets the minimum number of bytes to process for socket output
43093 operations. Non-blocking output operations shall process no data if flow
43094 control does not allow the smaller of the send low water mark value or
43095 the entire request to be processed. This option takes an **int** value. Note
43096 that not all implementations allow this option to be set.

43097 SO_SNDTIMEO Sets the timeout value specifying the amount of time that an output
43098 function blocks because flow control prevents data from being sent. If a
43099 send operation has blocked for this time, it shall return with a partial
43100 count or with *errno* set to [EAGAIN] or [EWOULDBLOCK] if no data is
43101 sent. The default for this option is zero, which indicates that a send
43102 operation shall not time out. This option stores a **timeval** structure. Note
43103 that not all implementations allow this option to be set.

43104 For Boolean options, 0 indicates that the option is disabled and 1 indicates that the option is
43105 enabled.

43106 Options at other protocol levels vary in format and name.

43107 **RETURN VALUE**

43108 Upon successful completion, *setsockopt*( ) shall return 0. Otherwise, –1 shall be returned and
43109 *errno* set to indicate the error.

43110 **ERRORS**

43111 The *setsockopt*( ) function shall fail if:

43112 [EBADF] The *socket* argument is not a valid file descriptor.

[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.

[ENOPROTOOPT] The option is not supported by the protocol.

[ENOTSOCK] The *socket* argument does not refer to a socket.

The *setsockopt*( ) function may fail if:

[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 60), *bind*( ), *endprotoent*( ), *getsockopt*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<netinet/in.h>**, **<sys/socket.h>**

CHANGE HISTORY

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

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/125 is applied, updating the SO_LINGER option in the DESCRIPTION to refer to the calling thread rather than the process.

43145 **NAME**

43146 setstate — switch pseudo-random number generator state arrays

43147 **SYNOPSIS**

43148 XSI
```
#include <stdlib.h>

char *setstate(const char *state);
```

43149

43150 **DESCRIPTION**

43151 Refer to *initstate( )*.

NAME

setuid — set user ID

SYNOPSIS

```
#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 the 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 real user ID, effective user ID, and saved set-user ID of the calling process 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 scope of IEEE Std 1003.1-200x. 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 1003.1-200x 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 1003.1-200x. 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 1003.1-200x for privileged users), and *seteuid*( ) (which always sets just the effective user ID, like *setuid*( ) in IEEE Std 1003.1-200x 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 1003.1-200x provide a new mechanism using *seteuid*( ) in conjunction with a saved set-user-ID. Thus, all implementations with the new *seteuid*( ) mechanism will have a saved set-user-ID for each process, and most of the behavior controlled by _POSIX_SAVED_IDS has been changed to agree with the case where the option was defined. The *kill*( ) function is an exception. Implementors of the new *seteuid*( ) mechanism will generally be required to maintain compatibility with the older mechanisms previously supported by their systems. However, compatibility with this use of *setreuid*( ) and with the _POSIX_SAVED_IDS behavior of *kill*( ) is unfortunately complicated. If an implementation with a saved set-user-ID allows a process to use *setreuid*( ) to swap its real and effective user IDs, but were to leave the saved set-user-ID unmodified, the process would then have an effective user ID equal to the original real user ID, and both real and saved set-user-ID would be equal to the original effective user ID. In that state, the real user would be unable to kill the process, even though the effective user ID of the process matches that of the real user, if the *kill*( ) behavior of _POSIX_SAVED_IDS was used. This is obviously not acceptable. The alternative choice, which is used in at least one implementation, is to change the saved set-user-ID to the effective user ID during most calls to *setreuid*( ). The standard developers considered that alternative to be less correct than the 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 1003.1-200x, **<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.

43269 **NAME**

43270 setutxent — reset the user accounting database to the first entry

43271 **SYNOPSIS**

43272 XSI
```
#include <utmpx.h>

void setutxent(void);
```

43274 **DESCRIPTION**

43275 Refer to *endutxent( )*.

setvbuf( )

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

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 u

43316 **FUTURE DIRECTIONS**

43317 None.

43318 **SEE ALSO**

43319 Section 2.5 (on page 34), *fopen( )*, *setbuf( )*, the Base Definitions volume of IEEE Std 1003.1-200x,
43320 **<stdio.h>**

43321 **CHANGE HISTORY**

43322 First released in Issue 1. Derived from Issue 1 of the SVID.

43323 **Issue 6**

43324 Extensions beyond the ISO C standard are marked.

43325 The *setvbuf( )* prototype is updated for alignment with the ISO/IEC 9899: 1999 standard.

NAME

shm_open — open a shared memory object (**REALTIME**)

SYNOPSIS

SHM

```
#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, except that the interpretation of slash characters other than the leading slash character in *name* is implementation-defined, and that the length limits for the *name* argument are implementation-defined and need not be the same as the pathname limits {PATH_MAX} and {NAME_MAX}. 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.

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.

[EINTR] The *shm_open*( ) operation was interrupted by a signal.

[EINVAL] The *shm_open*( ) operation is not supported for the given name.

[EMFILE] All file descriptors available to the process are currently open.

[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.

The *shm_open*( ) function may fail if:

[ENAMETOOLONG]
The length of the *name* argument exceeds {_POSIX_PATH_MAX} on systems that do not support the XSI option or exceeds {_XOPEN_PATH_MAX} on XSI systems, or has a pathname component that is longer than {_POSIX_NAME_MAX} on systems that do not support the XSI option or longer than {_XOPEN_NAME_MAX} on XSI systems.

EXAMPLES

Creating and Mapping a Shared Memory Object

The following code segment demonstrates the use of *shm_open*( ) to create a shared memory object which is then sized using *ftruncate*( ) before being mapped into the process address space using *mmap*( ):

```
#include <unistd.h>
#include <sys/mman.h>
...

#define MAX_LEN 10000
struct region {        /* Defines "structure" of shared memory */
```



```
        int len;
        char buf[MAX_LEN];
    };
    struct region *rptr;
    int fd;

    /* Create shared memory object and set its size */

    fd = shm_open("/myregion", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
    if (fd == -1)
        /* Handle error */;

    if (ftruncate(fd, sizeof(struct region)) == -1)
        /* Handle error */;

    /* Map shared memory object */

    rptr = mmap(NULL, sizeof(struct region),
           PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (rptr == MAP_FAILED)
        /* Handle error */;

    /* Now we can refer to mapped region using fields of rptr;
       for example, rptr->len */
    ...
```

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 set the size of an object referenced by a file descriptor.

On implementations where memory objects are implemented using the existing file system, the *shm_open*( ) function may be implemented using a macro that invokes *open*( ), and the *shm_unlink*( ) function may be implemented using a macro that invokes *unlink*( ).

For implementations without a permanent file system, the definition of the name of the memory objects is allowed not to survive a system reboot. Note that this allows systems with a permanent file system to implement memory objects as data structures internal to the implementation as well.

On implementations that choose to implement memory objects using memory directly, a *shm_open*( ) followed by an *ftruncate*( ) and *close*( ) can be used to preallocate a shared memory area and to set the size of that preallocation. This may be necessary for systems without virtual memory hardware support in order to ensure that the memory is contiguous.

The set of valid open flags to *shm_open*( ) was restricted to O_RDONLY, O_RDWR, O_CREAT, and O_TRUNC because these could be easily implemented on most memory mapping systems. This volume of IEEE Std 1003.1-200x 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 1003.1-200x 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.

FUTURE DIRECTIONS

None.

SEE ALSO

close( ), *dup*( ), *exec*, *fcntl*( ), *mmap*( ), *shmat*( ), *shmctl*( ), *shmdt*( ), *shm_unlink*( ), *umask*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/126 is applied, adding the example to the EXAMPLES section.

Issue 7

Austin Group Interpretation 1003.1-2001 #077 is applied, clarifying the *name* argument and changing [ENAMETOOLONG] from a "shall fail" to a "may fail" error.

NAME

shm_unlink — remove a shared memory object (**REALTIME**)

SYNOPSIS

SHM
```
#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.

[ENOENT] The named shared memory object does not exist.

The *shm_unlink*( ) function may fail if:

[ENAMETOOLONG]
The length of the *name* argument exceeds {_POSIX_PATH_MAX} on systems
XSI that do not support the XSI option or exceeds {_XOPEN_PATH_MAX} on XSI systems, or has a pathname component that is longer than
XSI {_POSIX_NAME_MAX} on systems that do not support the XSI option or longer than {_XOPEN_NAME_MAX} on XSI systems. A call to *shm_unlink*( ) with a *name* argument that contains the same shared memory object name as was previously used in a successful *shm_open*( ) call shall not give an [ENAMETOOLONG] error.

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.

43549 **RATIONALE**

43550 None.

43551 **FUTURE DIRECTIONS**

43552 None.

43553 **SEE ALSO**

43554 *close( )*, *mmap( )*, *munmap( )*, *shmat( )*, *shmctl( )*, *shmdt( )*, *shm_open( )*, the Base Definitions volume
43555 of IEEE Std 1003.1-200x, **<sys/mman.h>**

43556 **CHANGE HISTORY**

43557 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

43558 **Issue 6**

43559 The *shm_unlink( )* function is marked as part of the Shared Memory Objects option.

43560 In the DESCRIPTION, text is added to clarify that reusing the same name after a *shm_unlink( )*
43561 will not attach to the old shared memory object.

43562 The [ENOSYS] error condition has been removed as stubs need not be provided if an
43563 implementation does not support the Shared Memory Objects option.

43564 **Issue 7**

43565 Austin Group Interpretation 1003.1-2001 #077 is applied, changing [ENAMETOOLONG] from a
43566 "shall fail" to a "may fail" error.

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 1003.1-200x, 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 40).

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_nattch* 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 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 40), *exec*, *exit( )*, *fork( )*, *shmctl( )*, *shmdt( )*, *shmget( )*, *shm_open( )*, *shm_unlink( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<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 1003.1-200x, 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 40).

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.cuid* or *shm_perm.uid* 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.cuid* or *shm_perm.uid* 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.cuid* or *shm_perm.uid* 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 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 40), *shmat*( ), *shmdt*( ), *shmget*( ), *shm_open*( ), *shm_unlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

NAME

shmdt — XSI shared memory detach operation

SYNOPSIS

XSI
```
#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 1003.1-200x, 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 40).

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 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 40), *exec*, *exit*( ), *fork*( ), *shmat*( ), *shmctl*( ), *shmget*( ), *shm_open*( ), *shm_unlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

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 1003.1-200x, 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 40).

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

[ENOENT] A shared memory identifier does not exist for the argument *key* and (*shmflg* &IPC_CREAT) is 0.

[ENOMEM] A shared memory identifier and associated shared memory segment shall be created, but the amount of available physical memory is not sufficient to fill the request.

[ENOSPC] A shared memory identifier is to be created, but the system-imposed limit on the maximum number of allowed shared memory identifiers system-wide would be exceeded.

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 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 40), *shmat( )*, *shmctl( )*, *shmdt( )*, *shm_open( )*, *shm_unlink( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

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*

43846 **CHANGE HISTORY**
43847 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

sigaction — examine and change a signal action

SYNOPSIS

CX
```
#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:

Member Type	Member Name	Description
void(*) (int)	*sa_handler*	Pointer to a signal-catching function or one of the macros SIG_IGN or SIG_DFL.
sigset_t	*sa_mask*	Additional set of signals to be blocked during execution of signal-catching function.
int	*sa_flags*	Special flags to affect behavior of signal.
void(*) (int, siginfo_t *, void *)	*sa_sigaction*	Pointer to a signal-catching function.

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, the *sa_sigaction* field specifies a signal-catching function.

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*:

XSI SA_NOCLDSTOP Do not generate SIGCHLD when children stop or stopped children continue.

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.

XSI 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.

Otherwise, the disposition of the signal shall not be modified on entry to the signal handler.

In addition, if this flag is set, *sigaction*( ) behaves as if the SA_NODEFER flag were also set.

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 an interruptible function which uses a timeout is restarted, the duration of the timeout following the restart is set to an unspecified value that does not exceed the original timeout value. 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:

```
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:

```
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 thread's 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 *si_code* member contains a code identifying the cause of the signal.

If the value of *si_code* is less than or equal to 0, then the signal was generated by a process and *si_pid* and *si_uid*, respectively, indicate the process ID and the real user ID of the sender. The **<signal.h>** header

description contains information about the signal-specific contents of the elements of the **siginfo_t** type.

SA_NOCLDWAIT If set, and *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 *wait*( ), *waitid*( ), and *waitpid*( ) shall fail and set *errno* to [ECHILD]. Otherwise, terminating child processes shall be transformed into zombie processes, unless SIGCHLD is set to SIG_IGN.

SA_NODEFER If set and *sig* is caught, *sig* shall not be added to the thread's signal mask on entry to the signal handler unless it is included in *sa_mask*. Otherwise, *sig* shall always be added to the thread's signal mask on entry to the signal handler.

When a signal is caught by a signal-catching function installed by *sigaction*( ), a new signal mask is calculated and installed for the duration of the signal-catching function (or until a call to either *sigprocmask*( ) or *sigsuspend*( ) is made). This mask is formed by taking the union of the current signal mask and the value of the *sa_mask* for the signal being delivered, and unless SA_NODEFER or SA_RESETHAND is set, 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 *sigaction*( )), until the SA_RESETHAND flag causes resetting of the handler, or until one of the *exec* functions is called.

If the previous action for *sig* had been established by *signal*( ), the values of the fields returned in the structure pointed to by *oact* are unspecified, and in particular *oact->sa_handler* is not necessarily the same value passed to *signal*( ). However, if a pointer to the same structure or a copy thereof is passed to a subsequent call to *sigaction*( ) via the *act* argument, handling of the signal shall be as if the original call to *signal*( ) were repeated.

If *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 *errno* set to [EINVAL].

If SA_SIGINFO is not set in *sa_flags*, then the disposition of subsequent occurrences of *sig* when it is already pending is implementation-defined; the signal-catching function shall be invoked with a single argument. If SA_SIGINFO is set in *sa_flags*, then subsequent occurrences of *sig* generated by *sigqueue*( ) or as a result of any signal-generating function that supports the specification of an application-defined value (when *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 *si_value* member of the **siginfo_t** structure.

The result of the use of *sigaction*( ) and a *sigwait*( ) function concurrently within a process on the same signal is unspecified.

RETURN VALUE

Upon successful completion, *sigaction*( ) shall return 0; otherwise, –1 shall be returned, *errno* shall be set to indicate the error, and no new signal-catching function shall be installed.

ERRORS

The *sigaction*( ) function shall fail if:

[EINVAL] The *sig* argument is not a valid signal number or an attempt is made to catch a signal that cannot be caught or ignore a signal that cannot be ignored.

[ENOTSUP] The SA_SIGINFO bit flag is set in the *sa_flags* field of the **sigaction** structure.

The *sigaction*( ) function may fail if:

[EINVAL] An attempt was made to set the action to SIG_DFL for a signal that cannot be caught or ignored (or both).

In addition, the *sigaction*( ) function may fail if the SA_SIGINFO flag is set in the *sa_flags* field of the **sigaction** structure for a signal not in the range SIGRTMIN to SIGRTMAX.

EXAMPLES

Establishing a Signal Handler

The following example demonstrates the use of *sigaction*( ) to establish a handler for the SIGINT signal.

```
#include <signal.h>

static void handler(int signum)
{
    /* Take appropriate actions for signal delivery */
}

int main()
{
    struct sigaction sa;

    sa.sa_handler = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART; /* Restart functions if
                                 interrupted by handler */
    if (sigaction(SIGINT, &sa, NULL) == -1)
        /* Handle error */;

    /* Further code */
}
```

APPLICATION USAGE

The *sigaction*( ) function supersedes the *signal*( ) function, and should be used in preference. In particular, *sigaction*( ) and *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 1003.1-200x, 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 *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 *semget*( ), *sem_init*( ), *sem_open*( ), and so on). Note in particular that even the "safe" functions may modify *errno*; the signal-catching function, if not executing as an independent thread, may want to save and restore its value. Naturally, the same principles apply to the reentrancy of application routines and asynchronous data access. Note that *longjmp*( ) and *siglongjmp*( ) are not in the list of reentrant functions. This is because the code

executing after *longjmp*( ) and *siglongjmp*( ) can call any unsafe functions with the same danger as calling those unsafe functions directly from the signal handler. Applications that use *longjmp*( ) and *siglongjmp*( ) from within signal handlers require rigorous protection in order to be portable. Many of the other functions that are excluded from the list are traditionally implemented using either *malloc*( ) or *free*( ) functions or the standard I/O library, both of which traditionally use data structures in a non-reentrant manner. Since any combination of different functions using a common data structure can cause reentrancy problems, this volume of IEEE Std 1003.1-200x does not define the behavior when any unsafe function is called in a signal handler that interrupts an unsafe function.

If the signal occurs other than as the result of calling *abort*( ), *kill*( ), or *raise*( ), the behavior is undefined if the signal handler calls any function in the standard library other than one of the functions listed in the table above or refers to any object with static storage duration other than by assigning a value to a static storage duration variable of type **volatile sig_atomic_t**. Furthermore, if such a call fails, the value of *errno* is unspecified.

Usually, the signal is executed on the stack that was in effect before the signal was delivered. An alternate stack may be specified to receive a subset of the signals being caught.

When the signal handler returns, the receiving thread resumes execution at the point it was interrupted unless the signal handler makes other arrangements. If *longjmp*( ) or *_longjmp*( ) is used to leave the signal handler, then the signal mask must be explicitly restored.

This volume of IEEE Std 1003.1-200x 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 1003.1-200x. The BSD declaration would be:

```
void handler(int sig, int code, struct sigcontext *scp,
    char *addr);
```

where *sig* is the signal number, *code* is additional information on certain signals, *scp* is a pointer to the **sigcontext** structure, and *addr* is additional address information. Much the same information is available in the objects pointed to by the second argument of the signal handler specified when SA_SIGINFO is set.

RATIONALE

Although this volume of IEEE Std 1003.1-200x 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 e84 7 Tw(ed shatab01(.)Tj-44.9170 Tc11 arg)Tj0 Tu

This volume of IEEE Std 1003.1-200x 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 1003.1-200x 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-catching function.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.4 (on page 28), *exec*, *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 1003.1-200x, **<signal.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 1003.1-200x.

Text describing functionality from the Realtime Signals Extension option is marked.

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- The [ENOTSUP] error condition is added.

The normative text 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.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/57 is applied, changing text in the table describing the **sigaction** structure.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/127 is applied, removing text from the DESCRIPTION duplicated later in the same section.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/128 is applied, updating the DESCRIPTION and APPLICATION USAGE sections. Changes are made to refer to the thread rather than the process.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/129 is applied, adding the example to the EXAMPLES section.

Issue 7

Austin Group Interpretations 1003.1-2001 #065 and #084 are applied, clarifying the role of the SA_NODEFER flag with respect to the signal mask, and clarifying the SA_RESTART flag for interrupted functions which use timeouts.

Austin Group Interpretation 1003.1-2001 #004 is applied.

Functionality relating to the Realtime Signals Extension option is moved to the Base.

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 1003.1-200x, **<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 normative text 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

XSI
```
#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 for the current thread. 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.

```
#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), *exec*, *sigaction*( ), *sigsetjmp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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 normative text 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.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/58 is applied, updating the first sentence to include "for the current thread".

NAME

sigdelset — delete a signal from a signal set

SYNOPSIS

CX
```
#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*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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

CX
```
#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 1003.1-200x 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 1003.1-200x. Although it is outside the scope of this volume of IEEE Std 1003.1-200x 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.

44347 **SEE ALSO**

44348 Section 2.4 (on page 28), *sigaction( )*, *sigaddset( )*, *sigdelset( )*, *sigfillset( )*, *sigismember( )*, *sigpending( )*,
44349 *sigprocmask( )*, *sigsuspend( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

44350 **CHANGE HISTORY**

44351 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

44352 **Issue 6**

44353 The SYNOPSIS is marked CX since the presence of this function in the **<signal.h>** header is an
44354 extension over the ISO C standard.

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 1003.1-200x 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 1428).

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 1003.1-200x, **<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

OB XSI
```
#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 signal mask of the calling process before executing the signal handler; when the signal handler returns, the system shall restore the signal mask of the calling process 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 signal mask of the calling process 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 signal mask of the calling process.

The *sighold*( ) function shall add *sig* to the signal mask of the calling process.

The *sigrelse*( ) function shall remove *sig* from the signal mask of the calling process.

The *sigignore*( ) function shall set the disposition of *sig* to SIG_IGN.

The *sigpause*( ) function shall remove *sig* from the signal mask of the calling process and suspend the calling process until a signal is received. The *sigpause*( ) function shall restore the signal mask of the process 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.

44427 **ERRORS**

44428 These functions shall fail if:

44429 [EINVAL] The *sig* argument is an illegal signal number.

44430 The *sigset*( ) and *sigignore*( ) functions shall fail if:

44431 [EINVAL] An attempt is made to catch a signal that cannot be caught, or to ignore a
44432 signal that cannot be ignored.

44433 **EXAMPLES**

44434 None.

44435 **APPLICATION USAGE**

44436 The *sigaction*( ) function provides a more comprehensive and reliable mechanism for controlling
44437 signals; new applications should use the *sigaction*( ) function instead of the obsolescent *sigset*( )
44438 function.

44439 The *sighold*( ) function, in conjunction with *sigrelse*( ) or *sigpause*( ), may be used to establish
44440 critical regions of code that require the delivery of a signal to be temporarily deferred. For
44441 broader portability, the *pthread_sigmask*( ) or *sigprocmask*( ) functions should be used instead of
44442 the obsolescent *sighold*( ) and *sigrelse*( ) functions.

44443 For broader portability, the *sigsuspend*( ) function should be used instead of the obsolescent
44444 *sigpause*( ) function.

44445 **RATIONALE**

44446 Each of these historic functions has a direct analog in the other functions which are required to
44447 be per-thread and thread-safe (aside from *sigprocmask*( ), which is replaced by *pthread_sigmask*( )).
44448 The *sigset*( ) function can be implemented as a simple wrapper for *sigaction*( ). The *sighold*( )
44449 function is equivalent to *sigprocmask*( ) or *pthread_sigmask*( ) with SIG_BLOCK set. The *sigignore*( )
44450 function is equivalent to *sigaction*( ) with SIG_IGN set. The *sigpause*( ) function is equivalent to
44451 *sigsuspend*( ). The *sigrelse*( ) function is equivalent to *sigprocmask*( ) or *pthread_sigmask*( ) with
44452 SIG_UNBLOCK set.

44453 **FUTURE DIRECTIONS**

44454 These functions may be removed in a future version.

44455 **SEE ALSO**

44456 Section 2.4 (on page 28), *exec*, *pause*( ), *pthread_sigmask*( ), *sigaction*( ), *signal*( ), *sigsuspend*( ),
44457 *waitid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

44458 **CHANGE HISTORY**

44459 First released in Issue 4, Version 2.

44460 **Issue 5**

44461 Moved from X/OPEN UNIX extension to BASE.

44462 The DESCRIPTION is updated to indicate that the *sigpause*( ) function restores the signal mask of
44463 the process to its original state before returning.

44464 The RETURN VALUE section is updated to indicate that the *sigpause*( ) function suspends
44465 execution of the process until a signal is received, whereupon it returns –1 and sets *errno* to
44466 [EINTR].

44467 **Issue 6**

44468 The normative text is updated to avoid use of the term "must" for application requirements.

44469 References to the *wait3*( ) function are removed.

44470 The XSI functions are split out into their own reference page.

Issue 7

SD5-XSH-ERN-113 and SD5-XSH-ERN-42 are applied, marking these functions obsolescent and updating the APPLICATION USAGE and RATIONALE sections.

NAME

siginterrupt — allow signals to interrupt functions

SYNOPSIS

OB 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:

```
int 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. Applications should use the *sigaction*( ) with the SA_RESTART flag instead of the obsolescent *siginterrupt*( ) function.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.4 (on page 28), *sigaction*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/59 is applied, correcting the declaration in the sample implementation given in the DESCRIPTION.

Issue 7

The *siginterrupt*( ) function is marked obsolescent.

NAME

sigismember — test for a signal in a signal set

SYNOPSIS

CX
```
#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 1003.1-200x, **<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

CX
```
#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 *sigsetjmp*( ) with a non-zero *savemask* argument.

RETURN VALUE

After *siglongjmp*( ) is completed, program execution shall continue as if the corresponding invocation of *sigsetjmp*( ) had just returned the value specified by *val*. The *siglongjmp*( ) function shall not cause *sigsetjmp*( ) 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 *setjmp*( ) or *longjmp*( ) and *sigsetjmp*( ) or *siglongjmp*( ) is only significant for programs which use *sigaction*( ), *sigprocmask*( ), or *sigsuspend*( ).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

longjmp( ), *setjmp*( ), *sigprocmask*( ), *sigsetjmp*( ), *sigsuspend*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<setjmp.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 *longjmp*( ).

The SYNOPSIS is marked CX since the presence of this function in the **<setjmp.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

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 1003.1-200x 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 a:

```
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*.

ERRORS

The *signal*( ) function shall fail if:

CX [EINVAL] The *sig* argument is not a valid signal number or an attempt is made to catch a signal that cannot be caught or ignore a signal that cannot be ignored.

The *signal*( ) function may fail if:

CX [EINVAL] An attempt was made to set the action to SIG_DFL for a signal that cannot be caught or ignored (or both).

EXAMPLES

None.

APPLICATION USAGE

The *sigaction*( ) function provides a more comprehensive and reliable mechanism for controlling signals; new applications should use *sigaction*( ) rather than *signal*( ).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.4 (on page 28), *exec*, *pause*( ), *sigaction*( ), *sigsuspend*( ), *waitid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

CHANGE HISTORY

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

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The DESCRIPTION is updated to indicate that the *sigpause*( ) function restores the signal mask of the process 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

Extensions beyond the ISO C standard are marked.

The normative text 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 *wait3*( ) function are removed.

The *sighold*( ), *sigignore*( ), *sigrelse*( ), and *sigset*( ) functions are split out onto their own reference page.

NAME

signbit — test sign

SYNOPSIS

```
#include <math.h>

int signbit(real-floating 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 1003.1-200x 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( ), *isfinite*( ), *isinf*( ), *isnan*( ), *isnormal*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

NAME

sigpause — remove a signal from the signal mask and suspend the thread

SYNOPSIS

OB 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

exec, *sigaddset*( ), *sigdelset*( ), *sigemptyset*( ), *sigfillset*( ), *sigismember*( ), *sigprocmask*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

44742 **NAME**

44743 sigprocmask — examine and change blocked signals

44744 **SYNOPSIS**

44745 CX
```
#include <signal.h>

int sigprocmask(int how, const sigset_t *restrict set,
    sigset_t *restrict oset);
```

44748 **DESCRIPTION**

44749 Refer to *pthread_sigmask( )*.

NAME

sigqueue — queue a signal to a process

SYNOPSIS

CX
```
#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.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

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 1003.1-200x 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 1003.1-200x 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.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.8.1 (on page 41), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

CHANGE HISTORY

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

Issue 6

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.

Issue 7

The *sigqueue*( ) function is moved from the Realtime Signals Extension option to the Base.

NAME

sigrelse, sigset — signal management

SYNOPSIS

OB XSI

```
#include <signal.h>

int sigrelse(int sig);
void (*sigset(int sig, void (*disp)(int)))(int);
```

DESCRIPTION

Refer to *sighold( )*.

NAME

sigsetjmp — set jump point for a non-local goto

SYNOPSIS

CX
```
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savemask);
```

DESCRIPTION

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.

RETURN VALUE

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.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

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.

RATIONALE

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 1003.1-200x, **<setjmp.h>**

CHANGE HISTORY

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

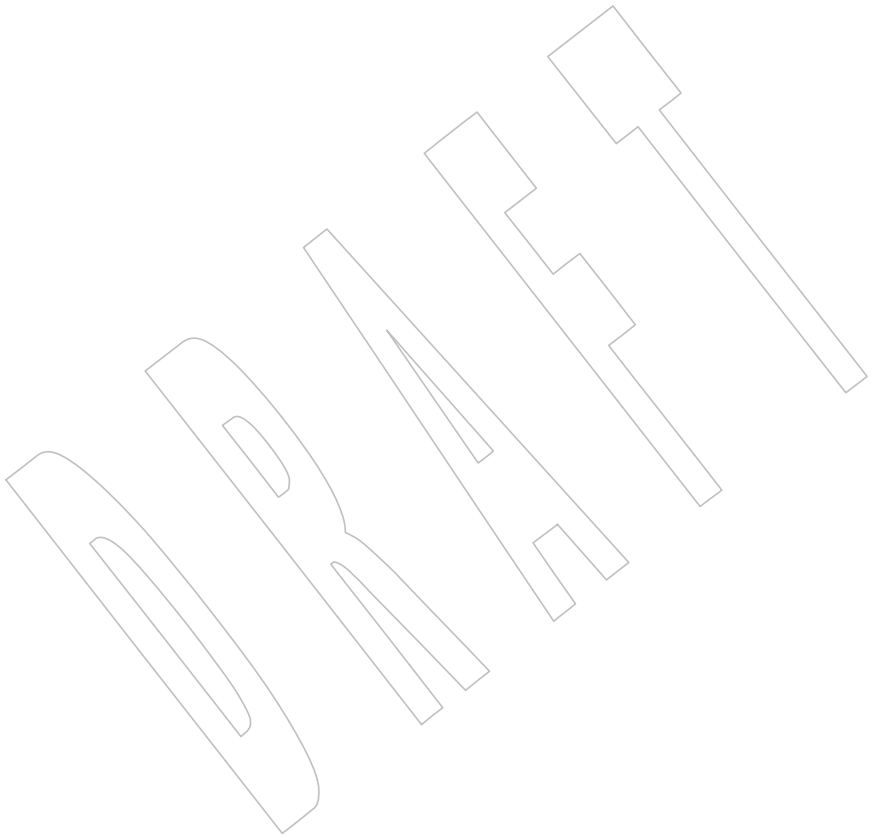
Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

The DESCRIPTION is reworded in terms of *setjmp*( ).

The SYNOPSIS is marked CX since the presence of this function in the **<setjmp.h>** header is an extension over the ISO C standard.



NAME

sigsuspend — wait for a signal

SYNOPSIS

CX
```
#include <signal.h>

int sigsuspend(const sigset_t *sigmask);
```

DESCRIPTION

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.

RETURN VALUE

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.

ERRORS

The *sigsuspend*( ) function shall fail if:

[EINTR] A signal is caught by the calling process and control is returned from the signal-catching function.

EXAMPLES

None.

APPLICATION USAGE

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.

RATIONALE

Code which wants to avoid the ambiguity of the signal mask for thread cancellation handlers can install an additional cancellation handler which resets the signal mask to the expected value.

```
void cleanup(void *arg)
{
    sigset_t *ss = (sigset_t *) arg;
    pthread_sigmask(SIG_SETMASK, ss, NULL);
}

int call_sigsuspend(const sigset_t *mask)
{
    sigset_t oldmask;
    int result;
    pthread_sigmask(SIG_SETMASK, NULL, &oldmask);
```



```
        pthread_cleanup_push(cleanup, &oldmask);
        result = sigsuspend(sigmask);
        pthread_cleanup_pop(0);
        return result;
    }
```

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.4 (on page 28), *pause*( ), *sigaction*( ), *sigaddset*( ), *sigdelset*( ), *sigemptyset*( ), *sigfillset*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.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 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.

Text in the APPLICATION USAGE section has been replaced.

The SYNOPSIS is marked CX since the presence of this function in the **<signal.h>** header is an extension over the ISO C standard.

Issue 7

SD5-XSH-ERN-122 is applied, adding the example code in the RATIONALE.

NAME

sigtimedwait, sigwaitinfo — wait for queued signals

SYNOPSIS

CX
```
#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
MON 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:

[EINVAL] The *timeout* argument specified a *tv_nsec* value less than zero or greater than or equal to 1 000 million.

An implementation should only check for this error if no signal is pending in *set* and it is necessary to wait.

EXAMPLES

None.

APPLICATION USAGE

The *sigtimedwait*( ) function times out and returns an [EAGAIN] error. Application writers should note that this is inconsistent with other functions such as *pthread_cond_timedwait*( ) that return [ETIMEDOUT].

RATIONALE

Existing programming practice on realtime 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 largescale 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 realtime signal function mechanism defined in this volume of IEEE Std 1003.1-200x. 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 1003.1-200x therefore provides two signal-waiting functions—one that waits indefinitely and one with a timeout—as part of the overall realtime 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 cancellation 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 1003.1-200x. 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 *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 1003.1-200x, **<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

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.

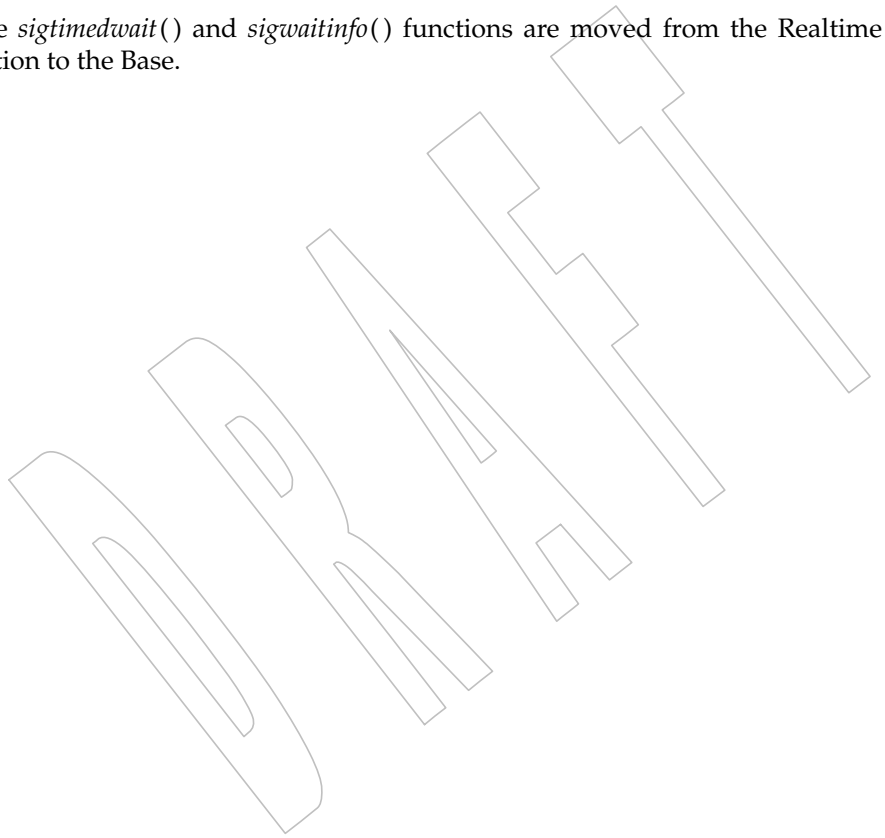
The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that the CLOCK_MONOTONIC clock, if supported, is used to measure timeout intervals.

The **restrict** keyword is added to the *sigtimedwait*( ) and *sigwaitinfo*( ) prototypes for alignment with the ISO/IEC 9899: 1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/130 is applied, restoring wording in the RETURN VALUE section to that in the original base document ("An implementation should only check for this error if no signal is pending in *set* and it is necessary to wait").

Issue 7

The *sigtimedwait*( ) and *sigwaitinfo*( ) functions are moved from the Realtime Signals Extension option to the Base.



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. If more than a single thread is blocked in *sigwait*( ) for a signal when that signal is generated for the process, it is unspecified which of the waiting threads returns from *sigwait*( ). If the signal is generated for a specific thread, as by *pthread_kill*( ), only that thread shall return.

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 1003.1-200x 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 1003.1-200x, **<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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/131 is applied, updating the DESCRIPTION section to state that if more than a single thread is blocked in *sigwait*( ), it is unspecified which of the waiting threads returns, and that if a signal is generated for a specific thread only that thread shall return.

Issue 7

Functionality relating to the Realtime Signals Extension option is moved to the Base.

45171 **NAME**

45172 sigwaitinfo — wait for queued signals

45173 **SYNOPSIS**

45174
```
#include <signal.h>
```

45175
```
int sigwaitinfo(const sigset_t *restrict set, siginfo_t *restrict info);
```

45176 **DESCRIPTION**

45177 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

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 1003.1-200x defers to the ISO C standard.

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 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

Taking the Sine of a 45-Degree Angle

```
#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 π 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

asin( ), *feclearexcept( )*, *fetestexcept( )*, *isnan( )*, the Base Definitions volume of IEEE Std 1003.1-200x, 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 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.

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

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

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 1003.1-200x 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 *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 sine of *x*.

If the result would cause an 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 type of the function.

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

If *x* is ±0 or ±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 (*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:

MX Range Error — The value *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

asinh( ), *cosh( )*, *feclearexcept( )*, *fetestexcept( )*, *isnan( )*, *tanh( )*, the Base Definitions volume of IEEE Std 1003.1-200x, 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 *sinhf( )* 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.

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

45305 **NAME**

45306 sinl — sine function

45307 **SYNOPSIS**

45308
```
#include <math.h>
```

45309
```
long double sinl(long double x);
```

45310 **DESCRIPTION**

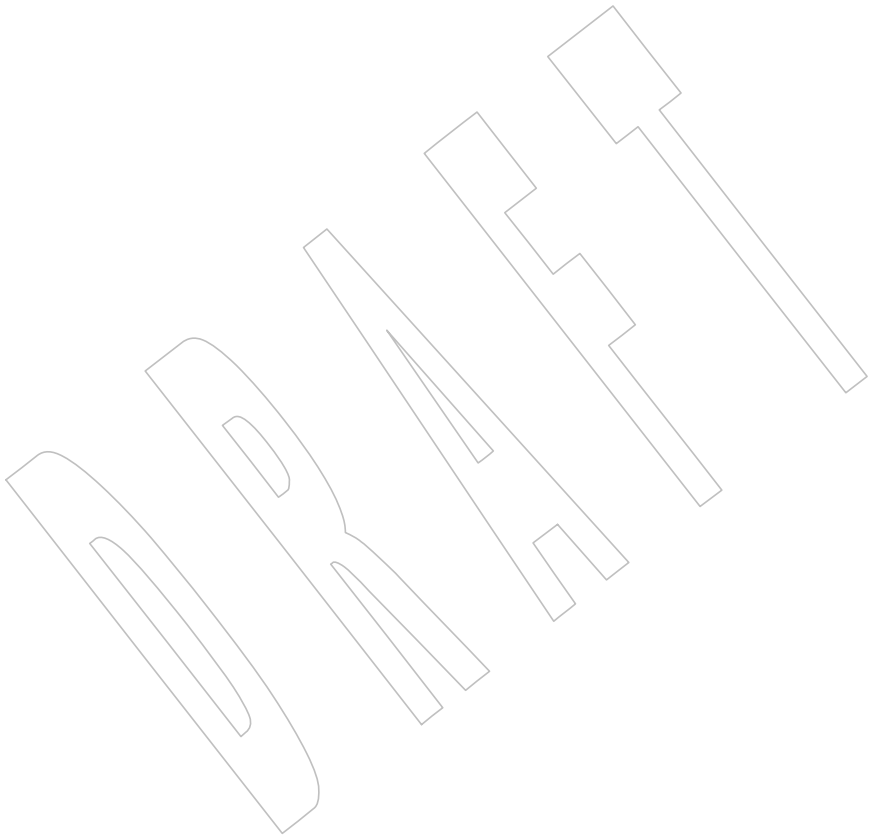
45311 Refer to *sin*( ).

NAME

sleep — suspend execution for an interval of time

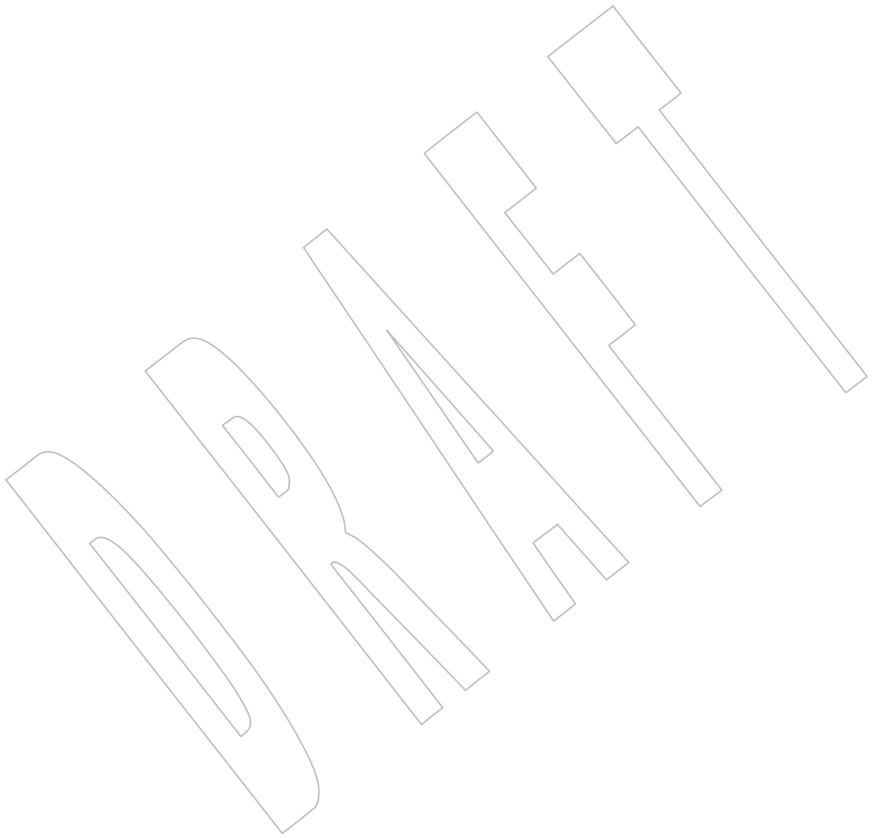
SYNOPSIS

```
#include <unistd.h>

unsigned sleep(unsigned seconds);
```



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*( ) c



45400 **NAME**

45401 snprintf — print formatted output

45402 **SYNOPSIS**

45403
```
#include <stdio.h>
```

45404
```
int snprintf(char *restrict s, size_t n,
    const char *restrict format, ...);
```

45406 **DESCRIPTION**

45407 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 1003.1-200x, 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 file associated with the *s* argument is not 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 62), Section 2.10.12 (on page 63), Section 2.10.14 (on page 63), 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)
{
```

```
        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 SIOCATMARK.

FUTURE DIRECTIONS

None.

SEE ALSO

pselect( ), *recv( )*, *recvmsg( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1g-2000.

Issue 7

SD5-XSH-ERN-100 is applied, correcting the definition of the [ENOTTY] error condition.

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.

45507 **ERRORS**

45508 The *socket*( ) function shall fail if:

45509 [EAFNOSUPPORT]

45510 The implementation does not support the specified address family.

45511 [EMFILE] All file descriptors available to the process are currently open.

45512 [ENFILE] No more file descriptors are available for the system.

45513 [EPROTONOSUPPORT]

45514 The protocol is not supported by the address family, or the protocol is not
45515 supported by the implementation.

45516 [EPROTOTYPE] The socket type is not supported by the protocol.

45517 The *socket*( ) function may fail if:

45518 [EACCES] The process does not have appropriate privileges.

45519 [ENOBUFS] Insufficient resources were available in the system to perform the operation.

45520 [ENOMEM] Insufficient memory was available to fulfill the request.

45521 **EXAMPLES**

45522 None.

45523 **APPLICATION USAGE**

45524 The documentation for specific address families specifies which protocols each address family
45525 supports. The documentation for specific protocols specifies which socket types each protocol
45526 supports.

45527 The application can determine whether an address family is supported by trying to create a
45528 socket with *domain* set to the protocol in question.

45529 **RATIONALE**

45530 None.

45531 **FUTURE DIRECTIONS**

45532 None.

45533 **SEE ALSO**

45534 *accept*( ), *bind*( ), *connect*( ), *getsockname*( ), *getsockopt*( ), *listen*( ), *recv*( ), *recvfrom*( ), *recvmsg*( ),
45535 *send*( ), *sendmsg*( ), *setsockopt*( ), *shutdown*( ), *socketpair*( ), the Base Definitions volume of
45536 IEEE Std 1003.1-200x, **<netinet/in.h>**, **<sys/socket.h>**

45537 **CHANGE HISTORY**

45538 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.

[EMFILE] All file descriptors available to the process are currently open.

[ENFILE] No more file descriptors are available for the system.

[EOPNOTSUPP] The specified protocol does not permit creation of socket pairs.

[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 *socketpair*( ) 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 *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 1003.1-200x, **<sys/socket.h>**

CHANGE HISTORY

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

sprintf( )

45609 **NAME**

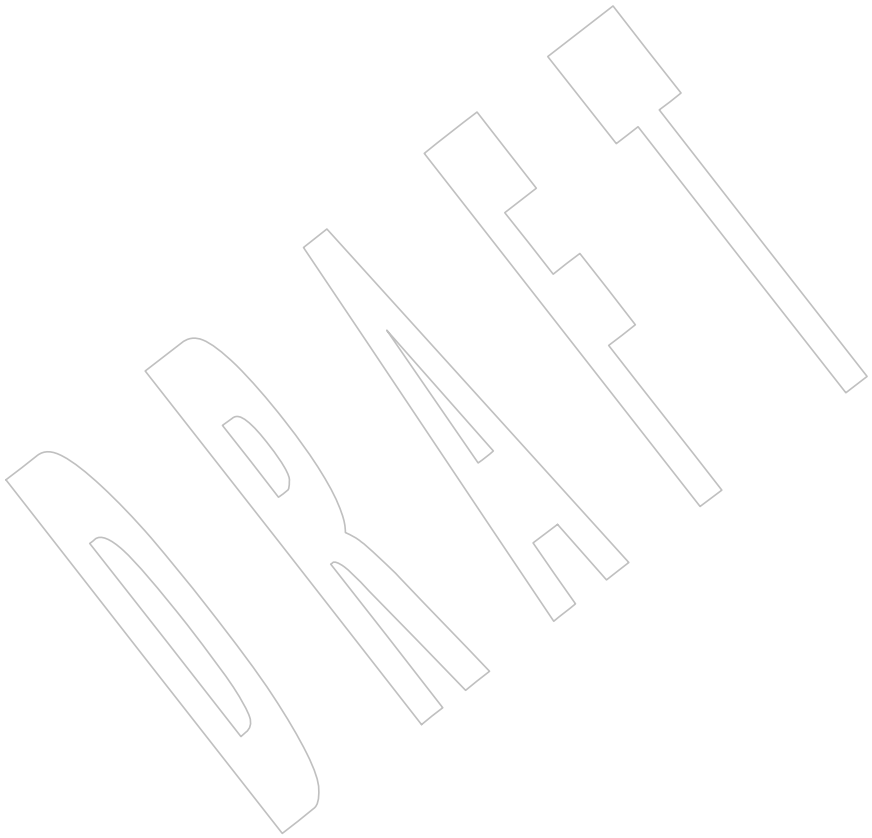
45610 sprintf — print formatted output

45611 **SYNOPSIS**

45612 ```
#include <stdio.h>
```

45613 ```
int sprintf(char *restrict s, const char *restrict format, ...);
```

45614 **DESCRIPTION**

45615 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

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 1003.1-200x 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 *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 square root of x.

MX For finite values of $x < -0$, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

MX 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:

MX Domain Error — The finite value of x is < -0, 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.

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 1003.1-200x, 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.

45676 **NAME**

45677 srand — pseudo-random number generator

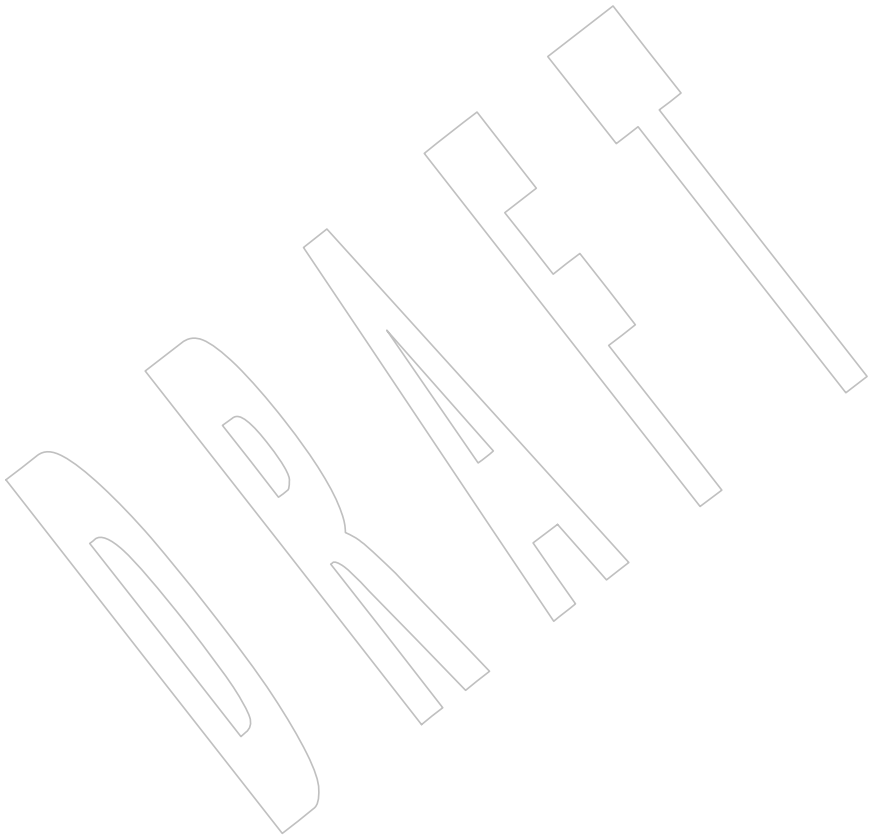
45678 **SYNOPSIS**

45679
```
#include <stdlib.h>
```

45680
```
void srand(unsigned seed);
```

45681 **DESCRIPTION**

45682 Refer to *rand( )*.



45683 **NAME**

45684 srand48 — seed the uniformly distributed double-precision pseudo-random number generator

45685 **SYNOPSIS**

45686 XSI
```
#include <stdlib.h>

void srand48(long seedval);
```

45688 **DESCRIPTION**

45689 Refer to *drand48( )*.

45690 **NAME**

45691 srandom — seed pseudo-random number generator

45692 **SYNOPSIS**

45693 XSI
```
#include <stdlib.h>

void srandom(unsigned seed);
```

45695 **DESCRIPTION**

45696 Refer to *initstate( )*.

45697 **NAME**

45698 sscanf — convert formatted input

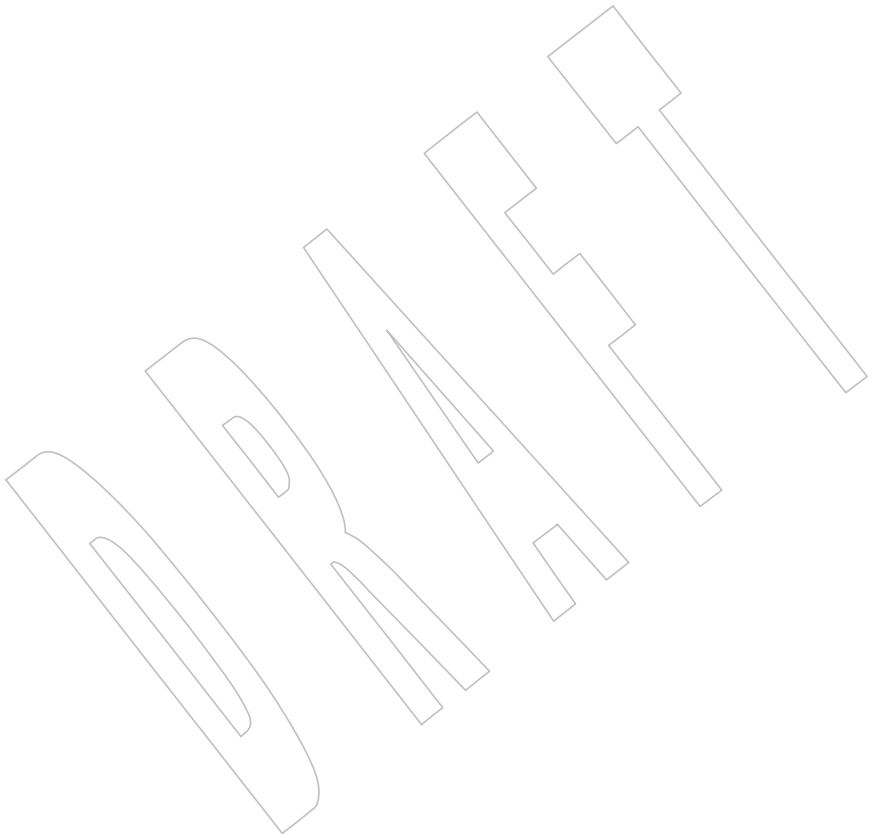
45699 **SYNOPSIS**

45700
```
#include <stdio.h>
```

45701
```
int sscanf(const char *restrict s, const char *restrict format, ...);
```

45702 **DESCRIPTION**

45703 Refer to *fscanf( )*.



45704 **NAME**

45705 stat — get file status

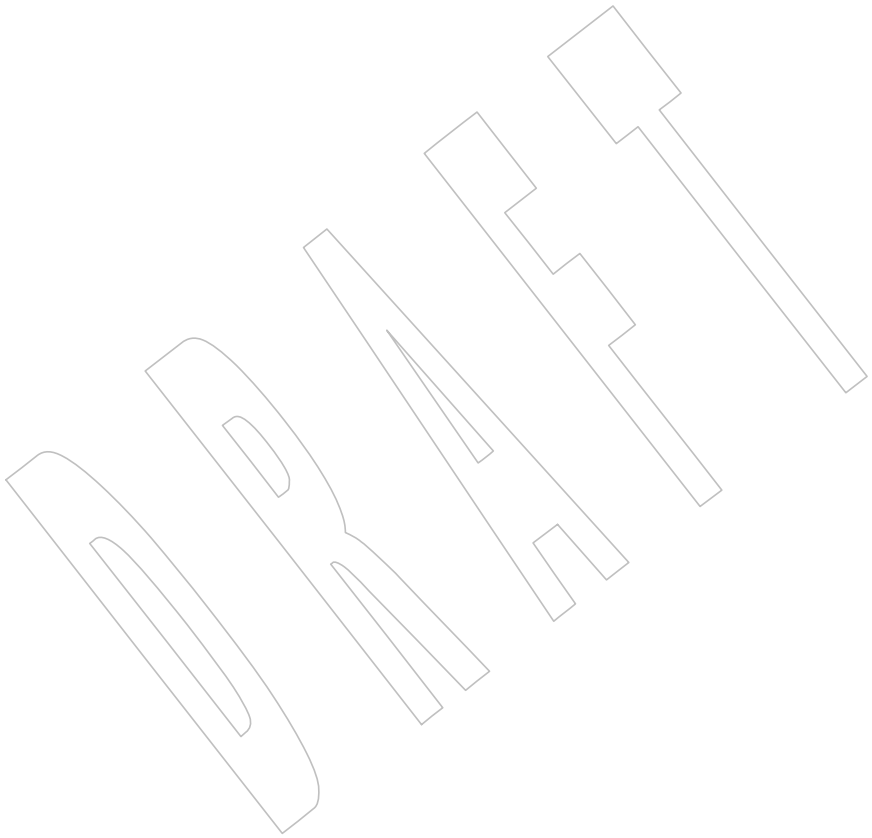
45706 **SYNOPSIS**

45707
```
#include <sys/stat.h>
```

45708
```
int stat(const char *restrict path, struct stat *restrict buf);
```

45709 **DESCRIPTION**

45710 Refer to *fstatat( )*.



45711 **NAME**

45712 statvfs — get file system information

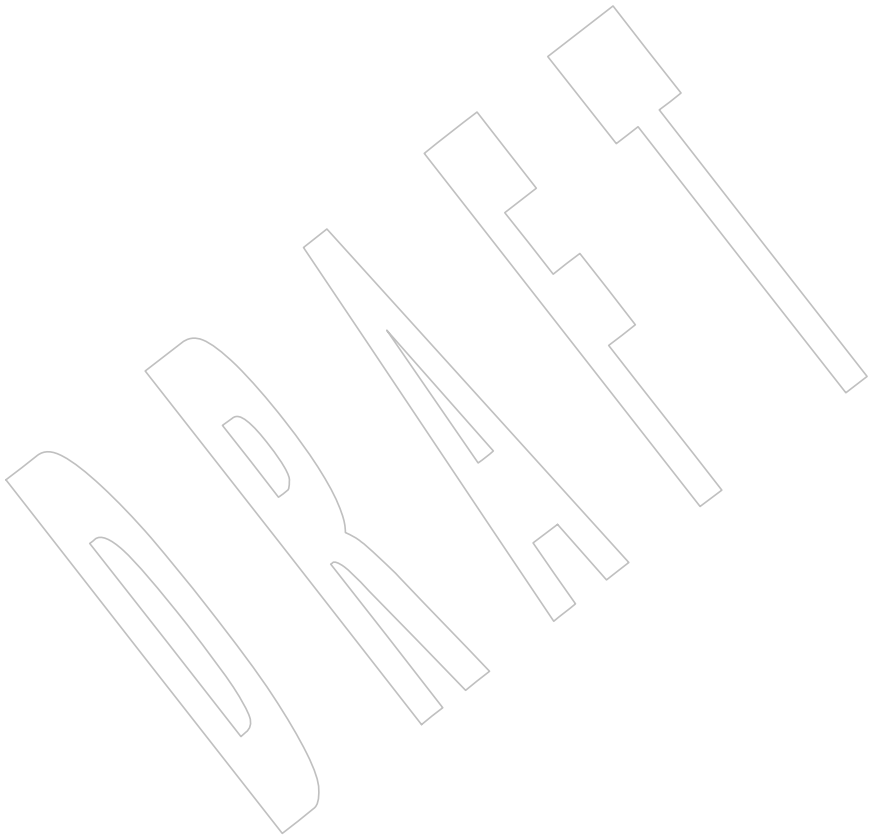
45713 **SYNOPSIS**

45714 `#include <sys/statvfs.h>`

45715 `int statvfs(const char *restrict path, struct statvfs *restrict buf);`

45716 **DESCRIPTION**

45717 Refer to *fstatvfs( )*.



45718 **NAME**

45719 stderr, stdin, stdout — standard I/O streams

45720 **SYNOPSIS**

45721
```
#include <stdio.h>
```

45722
```
extern FILE *stderr, *stdin, *stdout;
```

45723 **DESCRIPTION**

45724 CX The functionality described on this reference page is aligned with the ISO C standard. Any
45725 conflict between the requirements described here and the ISO C standard is unintentional. This
45726 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

45727 A file with associated buffering is called a *stream* and is declared to be a pointer to a defined type
45728 **FILE**. The *fopen*( ) function shall create certain descriptive data for a stream and return a pointer
45729 to designate the stream in all further transactions. Normally, there are three open streams with
45730 constant pointers declared in the **<stdio.h>** header and associated with the standard open files.

45731 At program start-up, three streams shall be predefined and need not be opened explicitly:
45732 *standard input* (for reading conventional input), *standard output* (for writing conventional output),
45733 and *standard error* (for writing diagnostic output). When opened, the standard error stream is not
45734 fully buffered; the standard input and standard output streams are fully buffered if and only if
45735 the stream can be determined not to refer to an interactive device.

45736 CX The following symbolic values in **<unistd.h>** define the file descriptors that shall be associated
45737 with the C-language *stdin*, *stdout*, and *stderr* when the application is started:

45738 STDIN_FILENO Standard input value, *stdin*. Its value is 0.

45739 STDOUT_FILENO Standard output value, *stdout*. Its value is 1.

45740 STDERR_FILENO Standard error value, *stderr*. Its value is 2.

45741 The *stderr* stream is expected to be open for reading and writing.

45742 **RETURN VALUE**

45743 None.

45744 **ERRORS**

45745 No errors are defined.

45746 **EXAMPLES**

45747 None.

45748 **APPLICATION USAGE**

45749 None.

45750 **RATIONALE**

45751 None.

45752 **FUTURE DIRECTIONS**

45753 None.

45754 **SEE ALSO**

45755 *fclose*( ), *feof*( ), *ferror*( ), *fileno*( ), *fopen*( ), *fread*( ), *fseek*( ), *getc*( ), *gets*( ), *popen*( ), *printf*( ), *putc*( ),
45756 *puts*( ), *read*( ), *scanf*( ), *setbuf*( ), *setvbuf*( ), *tmpfile*( ), *ungetc*( ), *vprintf*( ), the Base Definitions
45757 volume of IEEE Std 1003.1-200x, **<stdio.h>**, **<unistd.h>**

45758 **CHANGE HISTORY**

45759 First released in Issue 1.

45760 **Issue 6**

45761 Extensions beyond the ISO C standard are marked.

45762 A note that *stderr* is expected to be open for reading and writing is added to the DESCRIPTION.

45763 **NAME**

45764 stpcpy — copy a string and return a pointer to the end of the result

45765 **SYNOPSIS**

45766 CX
```
#include <string.h>

char *stpcpy(char *restrict s1, const char *restrict s2);
```

45768 **DESCRIPTION**

45769 Refer to *strcpy( )*.

45770 **NAME**

45771 stpncpy — copy fixed length string, returning a pointer to the array end

45772 **SYNOPSIS**

45773 CX
```
#include <string.h>
```

45774
```
char *stpncpy(char *restrict s1, const char *restrict s2, size_t size);
```

45775 **DESCRIPTION**

45776 Refer to *strncpy( )*.

NAME

strcasecmp, strcasecmp_l, strncasecmp, strncasecmp_l — case-insensitive string comparisons

SYNOPSIS

```
#include <strings.h>

int strcasecmp(const char *s1, const char *s2);
int strcasecmp_l(const char *s1, const char *s2,
    locale_t locale);
int strncasecmp(const char *s1, const char *s2, size_t n);
int strncasecmp_l(const char *s1, const char *s2,
    size_t n, locale_t locale);
```

DESCRIPTION

The *strcasecmp*( ) and *strcasecmp_l*( ) functions shall compare, while ignoring differences in case, the string pointed to by *s1* to the string pointed to by *s2*. The *strncasecmp*( ) and *strncasecmp_l*( ) functions 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*.

The *strcasecmp*( ) and *strncasecmp*( ) functions use the current locale of the process to determine the case of the characters.

The *strcasecmp_l*( ) and *strncasecmp_l*( ) functions use the locale represented by *locale* to determine the case of the characters.

When the *LC_CTYPE* category of the current locale is from the POSIX locale, *strcasecmp*( ) and *strncasecmp*( ) shall behave as if the strings had been converted to lowercase and then a byte comparison performed. Otherwise, the results are unspecified.

RETURN VALUE

Upon completion, *strcasecmp*( ) and *strcasecmp_l*( ) 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*( ) and *strncasecmp_l*( ) 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

The *strcasecmp_l*( ) and *strncasecmp_l*( ) functions may fail if:

[EINVAL] *locale* is not a valid locale object handle.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

45818 **SEE ALSO**

45819 *wcscasecmp( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<strings.h>**

45820 **CHANGE HISTORY**

45821 First released in Issue 4, Version 2.

45822 **Issue 5**

45823 Moved from X/OPEN UNIX extension to BASE.

45824 **Issue 7**

45825 The *strcasecmp*( ) and *strncasecmp*( ) functions are moved from the XSI option to the Base.

45826 The *strcasecmp_l*( ) and *strncasecmp_l*( ) functions are added from The Open Group Technical
45827 Standard, 2006, Extended API Set Part 4.

45828 **NAME**

45829 strcat — concatenate two strings

45830 **SYNOPSIS**

45831 ```
#include <string.h>

char *strcat(char *restrict s1, const char *restrict s2);
```

45833 **DESCRIPTION**

45834 CX The functionality described on this reference page is aligned with the ISO C standard. Any
45835 conflict between the requirements described here and the ISO C standard is unintentional. This
45836 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

45837 The *strcat*( ) function shall append a copy of the string pointed to by *s2* (including the
45838 terminating NUL character) to the end of the string pointed to by *s1*. The initial byte of *s2*
45839 overwrites the NUL character at the end of *s1*. If copying takes place between objects that
45840 overlap, the behavior is undefined.

45841 **RETURN VALUE**

45842 The *strcat*( ) function shall return *s1*; no return value is reserved to indicate an error.

45843 **ERRORS**

45844 No errors are defined.

45845 **EXAMPLES**

45846 None.

45847 **APPLICATION USAGE**

45848 This issue is aligned with the ISO C standard; this does not affect compatibility with XPG3
45849 applications. Reliable error detection by this function was never guaranteed.

45850 **RATIONALE**

45851 None.

45852 **FUTURE DIRECTIONS**

45853 None.

45854 **SEE ALSO**

45855 *strncat*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

45856 **CHANGE HISTORY**

45857 First released in Issue 1. Derived from Issue 1 of the SVID.

45858 **Issue 6**

45859 The *strcat*( ) prototype is updated for alignment with the ISO/IEC 9899: 1999 standard.

  
```

45860 **NAME**

45861 strchr — string scanning operation

45862 **SYNOPSIS**

45863
```
#include <string.h>
```

45864
```
char *strchr(const char *s, int c);
```

45865 **DESCRIPTION**

45866 CX The functionality described on this reference page is aligned with the ISO C standard. Any
45867 conflict between the requirements described here and the ISO C standard is unintentional. This
45868 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

45869 The *strchr*( ) function shall locate the first occurrence of *c* (converted to a **char**) in the string
45870 pointed to by *s*. The terminating NUL character is considered to be part of the string.

45871 **RETURN VALUE**

45872 Upon completion, *strchr*( ) shall return a pointer to the byte, or a null pointer if the byte was not
45873 found.

45874 **ERRORS**

45875 No errors are defined.

45876 **EXAMPLES**

45877 None.

45878 **APPLICATION USAGE**

45879 None.

45880 **RATIONALE**

45881 None.

45882 **FUTURE DIRECTIONS**

45883 None.

45884 **SEE ALSO**

45885 *strrchr*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

45886 **CHANGE HISTORY**

45887 First released in Issue 1. Derived from Issue 1 of the SVID.

45888 **Issue 6**

45889 Extensions beyond the ISO C standard are marked.

45890 **NAME**

45891 strcmp — compare two strings

45892 **SYNOPSIS**

45893
```
#include <string.h>
```

45894
```
int strcmp(const char *s1, const char *s2);
```

45895 **DESCRIPTION**

45896 CX The functionality described on this reference page is aligned with the ISO C standard. Any
45897 conflict between the requirements described here and the ISO C standard is unintentional. This
45898 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

45899 The *strcmp*( ) function shall compare the string pointed to by *s1* to the string pointed to by *s2*.

45900 The sign of a non-zero return value shall be determined by the sign of the difference between the
45901 values of the first pair of bytes (both interpreted as type **unsigned char**) that differ in the strings
45902 being compared.

45903 **RETURN VALUE**

45904 Upon completion, *strcmp*( ) shall return an integer greater than, equal to, or less than 0, if the
45905 string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2*,
45906 respectively.

45907 **ERRORS**

45908 No errors are defined.

45909 **EXAMPLES**

45910 **Checking a Password Entry**

45911 The following example compares the information read from standard input to the value of the
45912 name of the user entry. If the *strcmp*( ) function returns 0 (indicating a match), a further check
45913 will be made to see if the user entered the proper old password. The *crypt*( ) function shall
45914 encrypt the old password entered by the user, using the value of the encrypted password in the
45915 **passwd** structure as the salt. If this value matches the value of the encrypted **passwd** in the
45916 structure, the entered password *oldpasswd* is the correct user's password. Finally, the program
45917 encrypts the new password so that it can store the information in the **passwd** structure.

45918
```
#include <string.h>
#include <unistd.h>
#include <stdio.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");
            }
        }
        ...
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strncmp( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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

strcoll, strcoll_l — string comparison using collating information

SYNOPSIS

```
#include <string.h>

int strcoll(const char *s1, const char *s2);
CX int strcoll_l(const char *s1, const char *s2,
       locale_t locale);
```

DESCRIPTION

CX For *strcoll*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

CX The *strcoll*( ) and *strcoll_l*( ) functions 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 CX locale, or of the locale represented by *locale*, respectively.

CX The *strcoll*( ) and *strcoll_l*( ) functions 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 CX situations should set *errno* to 0, then call *strcoll*( ), or *strcoll_l*( ) 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 CX 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.

Upon successful completion, *strcoll_l*( ) 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 locale represented by *locale*. On error, *strcoll_l*( ) may set *errno*, but no return value is reserved to indicate an error.

ERRORS

These functions may fail if:

CX [EINVAL] The *s1* or *s2* arguments contain characters outside the domain of the collating sequence.

The *strcoll_l*( ) function may fail if:

CX [EINVAL] *locale* is not a valid locale object handle.

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

alphasort( ), *strcmp*( ), *strxfrm*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 3.

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:

- The [EINVAL] optional error condition is added.

An example is added.

Issue 7

The *strcoll_l*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

NAME

stpcpy, strcpy — copy a string and return a pointer to the end of the result

SYNOPSIS

```
#include <string.h>

char *stpcpy(char *restrict s1, const char *restrict s2);
char *strcpy(char *restrict s1, const char *restrict s2);
```

DESCRIPTION

For *strcpy*(): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *stpcpy*() and *strcpy*() functions shall copy the string pointed to by *s2* (including the terminating NUL character) into the array pointed to by *s1*.

If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE

The *stpcpy*() function shall return a pointer to the terminating NUL character copied into the *s1* buffer.

The *strcpy*() function shall return *s1*.

No return values are reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

Construction of a Multi-Part Message in a Single Buffer

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

int
main (void)
{
    char buffer [10];
    char *name = buffer;

    name = stpcpy (stpcpy (stpcpy (name, "ice"),"-"), "cream");
    puts (buffer);
    return 0;
}
```

Initializing a String

The following example copies the string "----------" into the *permstring* variable.

```
#include <string.h>
...
static char permstring[11];
...
strcpy(permstring, "----------");
...
```


46063 **Storing a Key and Data**

46064 The following example allocates space for a key using *malloc*( ) then uses *strcpy*( ) to place the
46065 key there. Then it allocates space for data using *malloc*( ), and uses *strcpy*( ) to place data there.
46066 (The user-defined function *dbfree*( ) frees memory previously allocated to an array of type **struct**
46067 **element** *.)

```
46068 #include <string.h>
46069 #include <stdlib.h>
46070 #include <stdio.h>
46071 ...
46072 /* Structure used to read data and store it. */
46073 struct element {
46074     char *key;
46075     char *data;
46076 };

46077 struct element *tbl, *curtbl;
46078 char *key, *data;
46079 int count;
46080 ...
46081 void dbfree(struct element *, int);
46082 ...
46083 if ((curtbl->key = malloc(strlen(key) + 1)) == NULL) {
46084     perror("malloc"); dbfree(tbl, count); return NULL;
46085 }
46086 strcpy(curtbl->key, key);

46087 if ((curtbl->data = malloc(strlen(data) + 1)) == NULL) {
46088     perror("malloc"); free(curtbl->key); dbfree(tbl, count); return NULL;
46089 }
46090 strcpy(curtbl->data, data);
46091 ...
```

46092 **APPLICATION USAGE**

46093 Character movement is performed differently in different implementations. Thus, overlapping
46094 moves may yield surprises.

46095 This issue is aligned with the ISO C standard; this does not affect compatibility with XPG3
46096 applications. Reliable error detection by this function was never guaranteed.

46097 **RATIONALE**

46098 None.

46099 **FUTURE DIRECTIONS**

46100 None.

46101 **SEE ALSO**

46102 *strncpy*( ), *wcscpy*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

46103 **CHANGE HISTORY**

46104 First released in Issue 1. Derived from Issue 1 of the SVID.

46105 **Issue 6**

46106 The *strcpy*( ) prototype is updated for alignment with the ISO/IEC 9899: 1999 standard.

Issue 7

The *stpcpy*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 1.

NAME

strcspn — get the length of a complementary substring

SYNOPSIS

```
#include <string.h>

size_t strcspn(const char *s1, const char *s2);
```

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 1003.1-200x defers to the ISO C standard.

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 1003.1-200x, **<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, strndup — duplicate a specific number of bytes from a string

SYNOPSIS

CX
```
#include <string.h>

char *strdup(const char *s);
char *strndup(const char *s, size_t size);
```

DESCRIPTION

The *strdup*( ) function shall return a pointer to a new string, which is a duplicate of the string pointed to by *s*. The returned pointer can be passed to *free*( ). A null pointer is returned if the new string cannot be created.

The *strndup*( ) function shall be equivalent to the *strdup*( ) function, duplicating the provided *s* in a new block of memory allocated as if by using *malloc*( ), with the exception being that *strndup*( ) copies at most *size* plus one bytes into the newly allocated memory, terminating the new string with a NUL character. If the length of *s* is larger than *size*, only *size* bytes shall be duplicated. If *size* is larger than the length of *s*, all bytes in *s* shall be copied into the new memory buffer, including the terminating NUL character. The newly created string shall always be properly terminated.

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.

Upon successful completion, the *strndup*( ) function shall return a pointer to the newly allocated memory containing the duplicated string. Otherwise, it shall return a null pointer and set *errno* to indicate the error.

ERRORS

These functions shall fail if:

[ENOMEM] Storage space available is insufficient.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

free( ), *malloc*( ), *wcsdup*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 7

Austin Group Interpretation 1003.1-2001 #044 is applied, changing the "may fail" [ENOMEM] error to become a "shall fail" error.

The *strdup*( ) function is moved from the XSI option to the Base.

The *strndup*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 1.

NAME

strerror, strerror_l, strerror_r — get error message string

SYNOPSIS

```
#include <string.h>

char *strerror(int errnum);
CX char *strerror_l(int errnum, locale_t locale);
int strerror_r(int errnum, char *strerrbuf, size_t buflen);
```

DESCRIPTION

CX For *strerror*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x 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. The string may be overwritten by a subsequent call to *strerror*( ) CX or *perror*( ).

CX The string may be overwritten by a subsquent call to *strerror_l*( ) in the same thread.

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 1003.1-200x calls *strerror*( ).

CX The *strerror*( ) and *strerror_l*( ) functions 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 thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

The *strerror_l*( ) function shall map the error number in *errnum* to a locale-dependent error message string in the locale represented by *locale* and shall return a pointer to it.

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 completion, whether successful or not, *strerror*( ) shall return a pointer to the generated CX message string. On error *errno* may be set, but no return value is reserved to indicate an error.

Upon successful completion, *strerror_l*( ) shall return a pointer to the generated message string. If *errnum* is not a valid error number, *errno* may be set to [EINVAL], but a pointer to a message string shall still be returned. If any other error occurs, *errno* shall be set to indicate the error and a null pointer shall be returned.

Upon successful completion, *strerror_r*( ) shall return 0. Otherwise, an error number shall be returned to indicate the error.

46230 **ERRORS**

46231 These functions may fail if:

46232 CX [EINVAL] The value of *errnum* is not a valid error number.

46233 The *strerror_l*( ) function may fail if:

46234 CX [EINVAL] The *locale* argument is not a valid locale object handle.

46235 The *strerror_r*( ) function may fail if:

46236 CX [ERANGE] Insufficient storage was supplied via *strerrbuf* and *buflen* to contain the
46237 generated message string.

46238 **EXAMPLES**

46239 None.

46240 **APPLICATION USAGE**

46241 None.

46242 **RATIONALE**

46243 The *strerror_l*( ) function is required to be thread-safe, thereby eliminating the need for an
46244 equivalent to the *strerror_r*( ) function.

46245 **FUTURE DIRECTIONS**

46246 None.

46247 **SEE ALSO**

46248 *perror*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

46249 **CHANGE HISTORY**

46250 First released in Issue 3.

46251 **Issue 5**

46252 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

46253 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

46254 **Issue 6**

46255 Extensions beyond the ISO C standard are marked.

46256 The following new requirements on POSIX implementations derive from alignment with the
46257 Single UNIX Specification:

46258 • In the RETURN VALUE section, the fact that *errno* may be set is added.

46259 • The [EINVAL] optional error condition is added.

46260 The normative text is updated to avoid use of the term "must" for application requirements.

46261 The *strerror_r*( ) function is added in response to IEEE PASC Interpretation 1003.1c #39.

46262 The *strerror_r*( ) function is marked as part of the Thread-Safe Functions option.

46263 **Issue 7**

46264 Austin Group Interpretation 1003.1-2001 #072 is applied, updating the ERRORS section.

46265 The *strerror_l*( ) function is added from The Open Group Technical Standard, 2006, Extended API
46266 Set Part 4.

46267 The *strerror_r*( ) function is moved from the Thread-Safe Functions option to the Base.

NAME

strfmon, strfmon_l — convert monetary value to a string

SYNOPSIS

```
#include <monetary.h>

ssize_t strfmon(char *restrict s, size_t maxsize,
    const char *restrict format, ...);
ssize_t strfmon_l(char *restrict s, size_t maxsize,
    locale_t locale, 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

The *strfmon_l*( ) function shall be equivalent to the *strfmon*( ) function, except that the locale data used is from the locale represented by *locale*.

Flags

One or more of the following optional flags can be specified to control the conversion:

=*f* 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.

+ or (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.

46308 `!` Suppress the currency symbol from the output conversion.

46309 – Specify the alignment. If this flag is present the result of the conversion is left-justified
46310 (padded to the right) rather than right-justified. This flag shall be ignored unless a field
46311 width (see below) is specified.

46312 **Field Width**

46313 *w* A decimal digit string *w* specifying a minimum field width in bytes in which the result
46314 of the conversion is right-justified (or left-justified if the flag `'-'` is specified). The
46315 default is 0.

46316 **Left Precision**

46317 *#n* A `'#'` followed by a decimal digit string *n* specifying a maximum number of digits
46318 expected to be formatted to the left of the radix character. This option can be used to
46319 keep the formatted output from multiple calls to the *strfmon*( ) function aligned in the
46320 same columns. It can also be used to fill unused positions with a special character as in
46321 `"$***123.45"`. This option causes an amount to be formatted as if it has the number
46322 of digits specified by *n*. If more than *n* digit positions are required, this conversion
46323 specification is ignored. Digit positions in excess of those actually required are filled
46324 with the numeric fill character (see the *=f* flag above).

46325 If grouping has not been suppressed with the `'^'` flag, and it is defined for the current
46326 locale, grouping separators are inserted before the fill characters (if any) are added.
46327 Grouping separators are not applied to fill characters even if the fill character is a digit.

46328 To ensure alignment, any characters appearing before or after the number in the
46329 formatted output such as currency or sign symbols are padded as necessary with
46330 <space>s to make their positive and negative formats an equal length.

46331 **Right Precision**

46332 *.p* A period followed by a decimal digit string *p* specifying the number of digits after the
46333 radix character. If the value of the right precision *p* is 0, no radix character appears. If a
46334 right precision is not included, a default specified by the current locale is used. The
46335 amount being formatted is rounded to the specified number of digits prior to
46336 formatting.

46337 **Conversion Specifier Characters**

46338 The conversion specifier characters and their meanings are:

46339 `i` The **double** argument is formatted according to the locale's international currency
46340 format (for example, in the U.S.: USD 1,234.56). If the argument is ±Inf or NaN, the
46341 result of the conversion is unspecified.

46342 `n` The **double** argument is formatted according to the locale's national currency format
46343 (for example, in the U.S.: $1,234.56). If the argument is ±Inf or NaN, the result of the
46344 conversion is unspecified.

46345 `%` Convert to a `'%'`; no argument is converted. The entire conversion specification shall
46346 be %%.

Locale Information

The *LC_MONETARY* category of the locale of the process affects the behavior of this function including the monetary radix character (which may be different from the numeric radix character affected by the *LC_NUMERIC* category), the grouping separator, the currency symbols, and formats. The international currency symbol should be conformant with the ISO 4217: 2001 standard.

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*, these functions shall return the number of bytes placed into the array pointed to by *s*, not including the terminating NUL character. Otherwise, –1 shall be returned, the contents of the array are unspecified, and *errno* shall be set to indicate the error.

ERRORS

These functions shall fail if:

[E2BIG] Conversion stopped due to lack of space in the buffer.

The *strfmon_l*( ) function may fail if:

[EINVAL] *locale* is not a valid locale object.

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]`
	`[-$123.45]`
	`[$3,456.78]`
		Default formatting

`%11n`	`[    $123.45]`
	`[   -$123.45]`
	`[  $3,456.78]`
		Right align within an 11-character field

`%#5n`	`[ $   123.45]`
	`[-$   123.45]`
	`[ $ 3,456.78]`
		Aligned columns for values up to 99 999

`%=*#5n`	`[ $***123.45]`
	`[-$***123.45]`
	`[ $*3,456.78]`
		Specify a fill character

`%=0#5n`	`[ $000123.45]`
	`[-$000123.45]`
	`[ $03,456.78]`
		Fill characters do not use grouping even if the fill character is a digit

`%^#5n`	`[ $  123.45]`
	`[-$  123.45]`
	`[ $ 3456.78]`
		Disable the grouping separator

`%^#5.0n`	`[ $  123]`
	`[-$  123]`
	`[ $ 3457]`
		Round off to whole units

`%^#5.4n`	`[ $  123.4500]`
	`[-$  123.4500]`
	`[ $ 3456.7810]`
		Increase the precision

`%(#5n`	`[ $   123.45 ]` `[($   123.45)]`	Use an alternative pos/neg style

```
                [ $ 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 1003.1-200x, **<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 normative text 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.

Issue 7

SD5-XSH-ERN-29 is applied, updating the examples for `%(#5n` and `%!(#5n`.

The *strfmon*( ) function is moved from the XSI option to the Base.

The *strfmon_l*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

NAME

strftime, strftime_l — 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);
CX size_t strftime_l(char *restrict s, size_t maxsize,
    const char *restrict format, const struct tm *restrict timeptr,
    locale_t locale);
```

DESCRIPTION

CX For *strftime*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

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 NUL character) 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.

CX The *strftime_l*( ) function shall be equivalent to the *strftime*( ) function, except that the locale data used is from the locale represented by *locale*.

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 1003.1-200x, **<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*]

46472		%F	Equivalent to %Y-%m-%d (the ISO 8601:2000 standard date format). [*tm_year*, *tm_mon*, *tm_mday*]
46474		%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*]
46476		%G	Replaced by the week-based year (see below) as a decimal number (for example, 1977). [*tm_year*, *tm_wday*, *tm_yday*]
46478		%h	Equivalent to %b. [*tm_mon*]
46479		%H	Replaced by the hour (24-hour clock) as a decimal number [00,23]. [*tm_hour*]
46480		%I	Replaced by the hour (12-hour clock) as a decimal number [01,12]. [*tm_hour*]
46481		%j	Replaced by the day of the year as a decimal number [001,366]. [*tm_yday*]
46482		%m	Replaced by the month as a decimal number [01,12]. [*tm_mon*]
46483		%M	Replaced by the minute as a decimal number [00,59]. [*tm_min*]
46484		%n	Replaced by a <newline>.
46485		%p	Replaced by the locale's equivalent of either a.m. or p.m. [*tm_hour*]
46486	CX	%r	Replaced by the time in a.m. and p.m. notation; in the POSIX locale this shall be equivalent to %I:%M:%S %p. [*tm_hour*, *tm_min*, *tm_sec*]
46488		%R	Replaced by the time in 24-hour notation (%H:%M). [*tm_hour*, *tm_min*]
46489		%S	Replaced by the second as a decimal number [00,60]. [*tm_sec*]
46490		%t	Replaced by a <tab>.
46491		%T	Replaced by the time (%H:%M:%S). [*tm_hour*, *tm_min*, *tm_sec*]
46492		%u	Replaced by the weekday as a decimal number [1,7], with 1 representing Monday. [*tm_wday*]
46494		%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*]
46497		%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*]
46502		%w	Replaced by the weekday as a decimal number [0,6], with 0 representing Sunday. [*tm_wday*]
46504		%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*]
46507		%x	Replaced by the locale's appropriate date representation. (See the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**.)
46509		%X	Replaced by the locale's appropriate time representation. (See the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**.)
46511		%y	Replaced by the last two digits of the year as a decimal number [00,99]. [*tm_year*]

46512 %Y Replaced by the year as a decimal number (for example, 1997). [*tm_year*]

46513 %z Replaced by the offset from UTC in the ISO 8601:2000 standard format (+hhmm or
46514 –hhmm), or by no characters if no timezone is determinable. For example, "–0430"
46515 CX means 4 hours 30 minutes behind UTC (west of Greenwich). If *tm_isdst* is zero, the
46516 standard time offset is used. If *tm_isdst* is greater than zero, the daylight savings time
46517 offset is used. If *tm_isdst* is negative, no characters are returned. [*tm_isdst*]

46518 %Z Replaced by the timezone name or abbreviation, or by no bytes if no timezone
46519 information exists. [*tm_isdst*]

46520 %% Replaced by %.

46521 If a conversion specification does not correspond to any of the above, the behavior is undefined.

46522 CX If a **struct tm** broken-down time structure is created by *localtime*( ) or *localtime_r*( ), or modified
46523 by *mktime*( ), and the value of *TZ* is subsequently modified, the results of the %Z and %z
46524 *strftime*( ) conversion specifiers are undefined, when *strftime*( ) is called with such a broken-down
46525 time structure.

46526 If a **struct tm** broken-down time structure is created or modified by *gmtime*( ) or *gmtime_r*( ), it is
46527 unspecified whether the result of the %Z and %z conversion specifiers shall refer to UTC or the
46528 current local timezone, when *strftime*( ) is called with such a broken-down time structure.

46529 **Modified Conversion Specifiers**

46530 Some conversion specifiers can be modified by the E or O modifier characters to indicate that an
46531 alternative format or specification should be used rather than the one normally used by the
46532 unmodified conversion specifier. If the alternative format or specification does not exist for the
46533 current locale (see ERA in the Base Definitions volume of IEEE Std 1003.1-200x, Section 7.3.5,
46534 LC_TIME), the behavior shall be as if the unmodified conversion specification were used.

46535 %Ec Replaced by the locale's alternative appropriate date and time representation.

46536 %EC Replaced by the name of the base year (period) in the locale's alternative
46537 representation.

46538 %Ex Replaced by the locale's alternative date representation.

46539 %EX Replaced by the locale's alternative time representation.

46540 %Ey Replaced by the offset from %EC (year only) in the locale's alternative representation.

46541 %EY Replaced by the full alternative year representation.

46542 %Od Replaced by the day of the month, using the locale's alternative numeric symbols, filled
46543 as needed with leading zeros if there is any alternative symbol for zero; otherwise, with
46544 leading spaces.

46545 %Oe Replaced by the day of the month, using the locale's alternative numeric symbols, filled
46546 as needed with leading spaces.

46547 %OH Replaced by the hour (24-hour clock) using the locale's alternative numeric symbols.

46548 %OI Replaced by the hour (12-hour clock) using the locale's alternative numeric symbols.

46549 %Om Replaced by the month using the locale's alternative numeric symbols.

46550 %OM Replaced by the minutes using the locale's alternative numeric symbols.

46551 %OS Replaced by the seconds using the locale's alternative numeric symbols.

46552 %Ou Replaced by the weekday as a number in the locale's alternative representation
46553 (Monday=1).

%OU Replaced by the week number of the year (Sunday as the first day of the week, rules corresponding to %U) using the locale's alternative numeric symbols.

%OV Replaced by the week number of the year (Monday as the first day of the week, rules corresponding to %V) using the locale's alternative numeric symbols.

%Ow Replaced by the number of the weekday (Sunday=0) using the locale's alternative numeric symbols.

%OW Replaced by the week number of the year (Monday as the first day of the week) using the locale's alternative numeric symbols.

%Oy Replaced by the year (offset from %C) using the locale's alternative numeric symbols.

%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*, these functions shall return the number of bytes placed into the array pointed to by *s*, not including the terminating NUL character. Otherwise, 0 shall be returned and the contents of the array are unspecified.

ERRORS

The *strftime_l*( ) function may fail if:

CX [EINVAL] *locale* is not a valid locale object handle.

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*.

```
#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 *nl_cxtime*( ) 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.

%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*( ), *strptime*( ), *time*( ), *tzset*( ), *uselocale*( ), *utime*( ), Base Definitions volume of IEEE Std 1003.1-200x, 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.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/60 is applied.

Issue 7

The *strftime_l*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

NAME

strlen, strnlen — get length of fixed size string

SYNOPSIS

```
#include <string.h>

size_t strlen(const char *s);
CX size_t strnlen(const char *s, size_t maxlen);
```

DESCRIPTION

CX For *strlen*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x 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 NUL character.

CX The *strnlen*( ) function shall compute the smaller of the number of bytes in the string to which *s* points, not including the terminating NUL character, or the value of the *maxlen* argument. The *strnlen*( ) function shall never examine more than *maxlen* bytes of the string pointed to by *s*.

RETURN VALUE

The *strlen*( ) function shall return the length of *s*; no return value shall be reserved to indicate an error.

CX The *strnlen*( ) function shall return an integer containing the smaller of either the length of the string pointed to by *s* or *maxlen*.

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;
...
```


46690 **APPLICATION USAGE**

46691 None.

46692 **RATIONALE**

46693 None.

46694 **FUTURE DIRECTIONS**

46695 None.

46696 **SEE ALSO**

46697 *wcslen*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

46698 **CHANGE HISTORY**

46699 First released in Issue 1. Derived from Issue 1 of the SVID.

46700 **Issue 5**

46701 The RETURN VALUE section is updated to indicate that *strlen*( ) returns the length of *s*, and not
46702 *s* itself as was previously stated.

46703 **Issue 7**

46704 The *strnlen*( ) function is added from The Open Group Technical Standard, 2006, Extended API
46705 Set Part 1.

NAME

strncasecmp, strncasecmp_l — case-insensitive string comparisons

SYNOPSIS

```
#include <strings.h>

int strncasecmp(const char *s1, const char *s2, size_t n);
int strncasecmp_l(const char *s1, const char *s2,
    size_t n, locale_t locale);
```

DESCRIPTION

Refer to *strcasecmp( )*.

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

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 1003.1-200x defers to the ISO C standard.

The *strncat*( ) function shall append not more than *n* bytes (a NUL character 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 NUL character at the end of *s1*. A terminating NUL character 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 1003.1-200x, **<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

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 1003.1-200x defers to the ISO C standard.

The *strncmp*( ) function shall compare not more than *n* bytes (bytes that follow a NUL character 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 1003.1-200x, **<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

stpncpy, strncpy — copy fixed length string, returning a pointer to the array end

SYNOPSIS

```
#include <string.h>

CX  char *stpncpy(char *restrict s1, const char *restrict s2, size_t n);
    char *strncpy(char *restrict s1, const char *restrict s2, size_t n);
```

DESCRIPTION

CX For *strncpy*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

CX The *stpncpy*( ) and *strncpy*( ) functions shall copy not more than *n* bytes (bytes that follow a NUL character are not copied) from the array pointed to by *s2* to the array pointed to by *s1*.

If the array pointed to by *s2* is a string that is shorter than *n* bytes, NUL characters shall be appended to the copy in the array pointed to by *s1*, until *n* bytes in all are written.

If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE

CX If a NUL character is written to the destination, the *stpncpy*( ) function shall return the address of the first such NUL character. Otherwise, it shall return *&s2*[*n*].

The *strncpy*( ) function shall return *s1*.

No return values are reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Applications must provide the space in *s1* for the *n* bytes to be transferred, as well as ensure that the *s2* and *s1* arrays do not overlap.

Character movement is performed differently in different implementations. Thus, overlapping moves may yield surprises.

If there is no NUL character byte in the first *n* bytes of the array pointed to by *s2*, the result is not null-terminated.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strcpy( ), *wcsncpy*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

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

Issue 6

The *strncpy*( ) prototype is updated for alignment with the ISO/IEC 9899: 1999 standard.

Issue 7

The *stpncpy*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 1.

46825 **NAME**

46826 strndup — duplicate a specific number of bytes from a string

46827 **SYNOPSIS**

46828 CX
```
#include <string.h>
```

46829
```
char *strndup(const char *s, size_t size);
```

46830 **DESCRIPTION**

46831 Refer to *strdup( )*.

46832 **NAME**

46833 strnlen — get length of fixed size string

46834 **SYNOPSIS**

46835 CX
```
#include <string.h>

size_t strnlen(const char *s, size_t maxlen);
```

46837 **DESCRIPTION**

46838 Refer to *strlen( )*.

NAME

strpbrk — scan a string for a byte

SYNOPSIS

```
#include <string.h>

char *strpbrk(const char *s1, const char *s2);
```

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 1003.1-200x defers to the ISO C standard.

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 1003.1-200x, **<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 *strptime*( ) 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 conversions are determined using the *LC_TIME* category of the current locale. The application shall ensure that there is white-space or other non-alphanumeric characters between any two conversion specifications. In the following list, where numeric ranges of values are given (represented by the pattern `[x,y]`), the value shall fall within the range given (both bounds being inclusive), and the number of characters scanned (excluding the one matching the next directive) shall be no more than the maximum number required to represent any value in the range without leading zeros. 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 shall be permitted but shall not be required.

`%d` The day of the month [01,31]; leading zeros shall be permitted but shall not be required.

`%D` The date as `%m/%d/%y`.

`%e` Equivalent to `%d`.

`%h` Equivalent to `%b`.

`%H` The hour (24-hour clock) [00,23]; leading zeros shall be permitted but shall not be required.

`%I` The hour (12-hour clock) [01,12]; leading zeros shall be permitted but shall not be required.

`%j` The day number of the year [001,366]; leading zeros shall be permitted but shall not be required.

`%m` The month number [01,12]; leading zeros shall be permitted but shall not be required.

%M The minute [00,59]; leading zeros shall be permitted but shall not be 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 **t_fmt_ampm** is not an empty string in the *LC_TIME* portion of the current locale; in the POSIX locale, this shall be equivalent to %I:%M:%S %p.

%R The time as %H:%M.

%S The seconds [00,60]; leading zeros shall be permitted but shall not be 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 shall be permitted but shall not be required.

%w The weekday as a decimal number [0,6], with 0 representing Sunday.

%W The week number of the year (Monday as the first day of the week) as a decimal number [00,53]; leading zeros shall be permitted but shall not be 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 1003.1-200x 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 shall be permitted but shall not be required.

46946 `%Oe` Equivalent to `%Od`.

46947 `%OH` The hour (24-hour clock) using the locale's alternative numeric symbols.

46948 `%OI` The hour (12-hour clock) using the locale's alternative numeric symbols.

46949 `%Om` The month using the locale's alternative numeric symbols.

46950 `%OM` The minutes using the locale's alternative numeric symbols.

46951 `%OS` The seconds using the locale's alternative numeric symbols.

46952 `%OU` The week number of the year (Sunday as the first day of the week) using the locale's
46953 alternative numeric symbols.

46954 `%Ow` The number of the weekday (Sunday=0) using the locale's alternative numeric
46955 symbols.

46956 `%OW` The week number of the year (Monday as the first day of the week) using the locale's
46957 alternative numeric symbols.

46958 `%Oy` The year (offset from `%C`) using the locale's alternative numeric symbols.

46959 A conversion specification composed of white-space characters is executed by scanning input up
46960 to the first character that is not white-space (which remains unscanned), or until no more
46961 characters can be scanned.

46962 A conversion specification that is an ordinary character is executed by scanning the next
46963 character from the buffer. If the character scanned from the buffer differs from the one
46964 comprising the directive, the directive fails, and the differing and subsequent characters remain
46965 unscanned.

46966 A series of conversion specifications composed of `%n`, `%t`, white-space characters, or any
46967 combination is executed by scanning up to the first character that is not white space (which
46968 remains unscanned), or until no more characters can be scanned.

46969 Any other conversion specification is executed by scanning characters until a character matching
46970 the next directive is scanned, or until no more characters can be scanned. These characters,
46971 except the one matching the next directive, are then compared to the locale values associated
46972 with the conversion specifier. If a match is found, values for the appropriate **tm** structure
46973 members are set to values corresponding to the locale information. Case is ignored when
46974 matching items in *buf* such as month or weekday names. If no match is found, *strptime*( ) fails
46975 and no more characters are scanned.

46976 **RETURN VALUE**

46977 Upon successful completion, *strptime*( ) shall return a pointer to the character following the last
46978 character parsed. Otherwise, a null pointer shall be returned.

46979 **ERRORS**

46980 No errors are defined.

46981 **EXAMPLES**

46982 **Convert a Data-Plus-Time String to Broken-Down Time and Then into Seconds**

46983 The following example demonstrates the use of *strptime*( ) to convert a string into broken-down
46984 time. The broken-down time is then converted into seconds since the Epoch using *mktime*( ).

```
46985 #include <time.h>
46986 ...
46987 struct tm tm;
46988 time_t t;
```



```
if (strptime("6 Dec 2001 12:33:45", "%d %b %Y %H:%M:%S", &tm) == NULL)
    /* Handle error */;

printf("year: %d; month: %d; day: %d;\n",
        tm.tm_year, tm.tm_mon, tm.tm_mday);
printf("hour: %d; minute: %d; second: %d\n",
        tm.tm_hour, tm.tm_min, tm.tm_sec);
printf("week day: %d; year day: %d\n", tm.tm_wday, tm.tm_yday);

tm.tm_isdst = –1;      /* Not set by strptime(); tells mktime()
                          to determine whether daylight saving time
                          is in effect */
t = mktime(&tm);
if (t == –1)
    /* Handle error */;
printf("seconds since the Epoch: %ld\n", (long) t);"
```

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*( ).

It should be noted that dates constructed by the *strftime*( ) function with the %Y or %C%y conversion specifiers may have values larger than 9999. If the *strptime*( ) function is used to read this value using %C%y, these values will be truncated to four digits.

Applications should use %Y (year including century) 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 1003.1-200x.

SEE ALSO

scanf( ), *strftime*( ), *time*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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 normative text 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

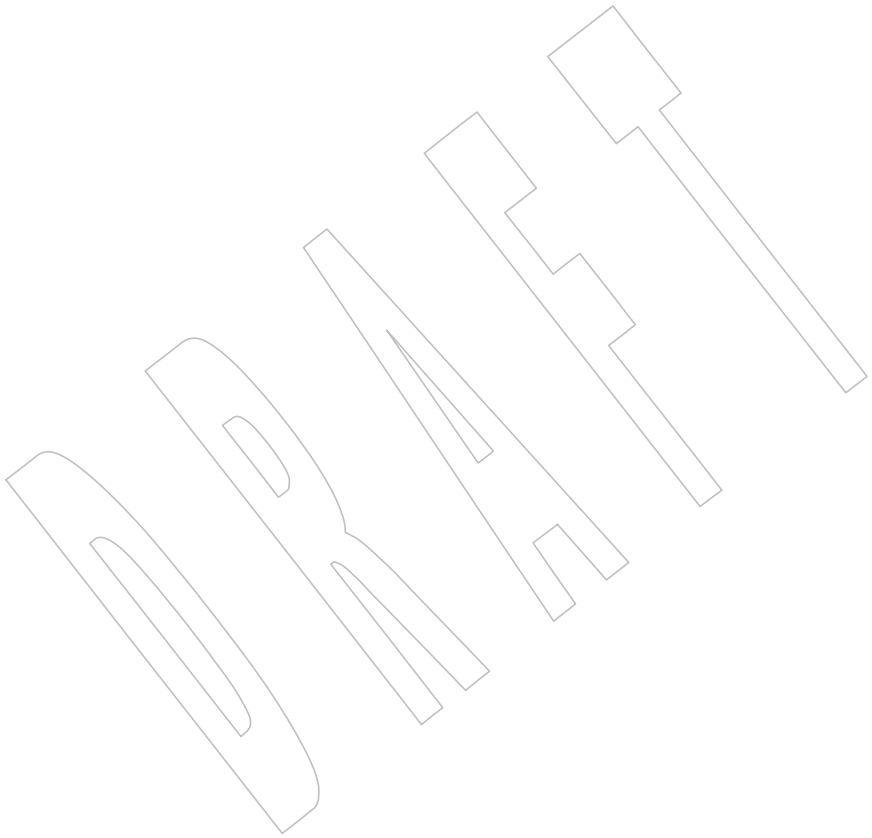
47034 specification'' for consistency with *strftime*( ).

47035 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/133 is applied, adding the example to the
47036 EXAMPLES section.

47037 **Issue 7**

47038 SD5-XSH-ERN-67 is applied, correcting the APPLICATION USAGE to remove the impression
47039 that %Y is four digit years.

47040 Austin Group Interpretation 1003.1-2001 #041 is applied, updating the DESCRIPTION and
47041 APPLICATION USAGE sections.



47042 **NAME**

47043 strrchr — string scanning operation

47044 **SYNOPSIS**

47045
```
#include <string.h>
```

47046
```
char *strrchr(const char *s, int c);
```

47047 **DESCRIPTION**

47048 CX The functionality described on this reference page is aligned with the ISO C standard. Any
47049 conflict between the requirements described here and the ISO C standard is unintentional. This
47050 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

47051 The *strrchr*( ) function shall locate the last occurrence of *c* (converted to a **char**) in the string
47052 pointed to by *s*. The terminating NUL character is considered to be part of the string.

47053 **RETURN VALUE**

47054 Upon successful completion, *strrchr*( ) shall return a pointer to the byte or a null pointer if *c* does
47055 not occur in the string.

47056 **ERRORS**

47057 No errors are defined.

47058 **EXAMPLES**

47059 **Finding the Base Name of a File**

47060 The following example uses *strrchr*( ) to get a pointer to the base name of a file. The *strrchr*( )
47061 function searches backwards through the name of the file to find the last `'/'` character in *name*.
47062 This pointer (plus one) will point to the base name of the file.

47063
```
#include <string.h>
...
const char *name;
char *basename;
...
basename = strrchr(name, '/') + 1;
...
```

47070 **APPLICATION USAGE**

47071 None.

47072 **RATIONALE**

47073 None.

47074 **FUTURE DIRECTIONS**

47075 None.

47076 **SEE ALSO**

47077 *strchr*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

47078 **CHANGE HISTORY**

47079 First released in Issue 1. Derived from Issue 1 of the SVID.

47080 **NAME**

47081 strsignal — get name of signal

47082 **SYNOPSIS**

47083 CX
```
#include <string.h>

char *strsignal(int signum);
```
47084

47085 **DESCRIPTION**

47086 The *strsignal*( ) function shall map the signal number in *signum* to an implementation-defined
47087 string and shall return a pointer to it. It shall use the same set of messages as the *psignal*( )
47088 function.

47089 The string pointed to shall not be modified by the application, but may be overwritten by a
47090 subsequent call to *strsignal*( ) or *setlocale*( ).

47091 The contents of the message strings returned by *strsignal*( ) should be determined by the setting
47092 of the *LC_MESSAGES* category in the current locale.

47093 The implementation shall behave as if no function defined in this standard calls *strsignal*( ).

47094 Since no return value is reserved to indicate an error, an application wishing to check for error
47095 situations should set *errno* to 0, then call *strsignal*( ), then check *errno*.

47096 The *strsignal*( ) function need not be reentrant. A function that is not required to be reentrant is
47097 not required to be thread-safe.

47098 **RETURN VALUE**

47099 Upon successful completion, *strsignal*( ) shall return a pointer to a string. Otherwise, if *signum* is
47100 not a valid signal number, the return value is unspecified.

47101 **ERRORS**

47102 No errors are defined.

47103 **EXAMPLES**

47104 None.

47105 **APPLICATION USAGE**

47106 None.

47107 **RATIONALE**

47108 If *signum* is not a valid signal number, some implementations return NULL, while for others the
47109 *strsignal*( ) function returns a pointer to a string containing an unspecified message denoting an
47110 unknown signal. This standard leaves this return value unspecified.

47111 **FUTURE DIRECTIONS**

47112 None.

47113 **SEE ALSO**

47114 *psiginfo*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

47115 **CHANGE HISTORY**

47116 First released in Issue 7.

47117 **NAME**

47118 strspn — get length of a substring

47119 **SYNOPSIS**

47120
```
#include <string.h>
```

47121
```
size_t strspn(const char *s1, const char *s2);
```

47122 **DESCRIPTION**

47123 CX The functionality described on this reference page is aligned with the ISO C standard. Any
47124 conflict between the requirements described here and the ISO C standard is unintentional. This
47125 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

47126 The *strspn*( ) function shall compute the length (in bytes) of the maximum initial segment of the
47127 string pointed to by *s1* which consists entirely of bytes from the string pointed to by *s2*.

47128 **RETURN VALUE**

47129 The *strspn*( ) function shall return the length of *s1*; no return value is reserved to indicate an
47130 error.

47131 **ERRORS**

47132 No errors are defined.

47133 **EXAMPLES**

47134 None.

47135 **APPLICATION USAGE**

47136 None.

47137 **RATIONALE**

47138 None.

47139 **FUTURE DIRECTIONS**

47140 None.

47141 **SEE ALSO**

47142 *strcspn*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

47143 **CHANGE HISTORY**

47144 First released in Issue 1. Derived from Issue 1 of the SVID.

47145 **Issue 5**

47146 The RETURN VALUE section is updated to indicate that *strspn*( ) returns the length of *s*, and not
47147 *s* itself as was previously stated.

47148 **NAME**

47149 strstr — find a substring

47150 **SYNOPSIS**

47151
```
#include <string.h>
```

47152
```
char *strstr(const char *s1, const char *s2);
```

47153 **DESCRIPTION**

47154 CX The functionality described on this reference page is aligned with the ISO C standard. Any
47155 conflict between the requirements described here and the ISO C standard is unintentional. This
47156 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

47157 The *strstr*( ) function shall locate the first occurrence in the string pointed to by *s1* of the
47158 sequence of bytes (excluding the terminating NUL character) in the string pointed to by *s2*.

47159 **RETURN VALUE**

47160 Upon successful completion, *strstr*( ) shall return a pointer to the located string or a null pointer
47161 if the string is not found.

47162 If *s2* points to a string with zero length, the function shall return *s1*.

47163 **ERRORS**

47164 No errors are defined.

47165 **EXAMPLES**

47166 None.

47167 **APPLICATION USAGE**

47168 None.

47169 **RATIONALE**

47170 None.

47171 **FUTURE DIRECTIONS**

47172 None.

47173 **SEE ALSO**

47174 *strchr*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

47175 **CHANGE HISTORY**

47176 First released in Issue 3. Included for alignment with the ANSI C standard.

47177 **NAME**

47178 strtod, strtof, strtold — convert a string to a double-precision number

47179 **SYNOPSIS**

47180
```
#include <stdlib.h>
```

47181
```
double strtod(const char *restrict nptr, char **restrict endptr);
```
47182
```
float strtof(const char *restrict nptr, char **restrict endptr);
```
47183
```
long double strtold(const char *restrict nptr, char **restrict endptr);
```

47184 **DESCRIPTION**

47185 CX The functionality described on this reference page is aligned with the ISO C standard. Any
47186 conflict between the requirements described here and the ISO C standard is unintentional. This
47187 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

47188 These functions shall convert the initial portion of the string pointed to by *nptr* to **double**, **float**,
47189 and **long double** representation, respectively. First, they decompose the input string into three
47190 parts:

47191 1. An initial, possibly empty, sequence of white-space characters (as specified by *isspace*( ))

47192 2. A subject sequence interpreted as a floating-point constant or representing infinity or
47193 NaN

47194 3. A final string of one or more unrecognized characters, including the terminating NUL
47195 character of the input string

47196 Then they shall attempt to convert the subject sequence to a floating-point number, and return
47197 the result.

47198 The expected form of the subject sequence is an optional plus or minus sign, then one of the
47199 following:

47200 • A non-empty sequence of decimal digits optionally containing a radix character; then an
47201 optional exponent part consisting of the character `'e'` or the character `'E'`, optionally
47202 followed by a `'+'` or `'−'` character, and then followed by one or more decimal digits

47203 • A 0x or 0X, then a non-empty sequence of hexadecimal digits optionally containing a radix
47204 character; then an optional binary exponent part consisting of the character `'p'` or the
47205 character `'P'`, optionally followed by a `'+'` or `'−'` character, and then followed by one or
47206 more decimal digits

47207 • One of INF or INFINITY, ignoring case

47208 • One of NAN or NAN(*n-char-sequence*$_{opt}$), ignoring case in the NAN part, where:

47209
```
n-char-sequence:
    digit
    nondigit
    n-char-sequence digit
    n-char-sequence nondigit
```

47214 The subject sequence is defined as the longest initial subsequence of the input string, starting
47215 with the first non-white-space character, that is of the expected form. The subject sequence
47216 contains no characters if the input string is not of the expected form.

47217 If the subject sequence has the expected form for a floating-point number, the sequence of
47218 characters starting with the first digit or the decimal-point character (whichever occurs first)
47219 shall be interpreted as a floating constant of the C language, except that the radix character shall
47220 be used in place of a period, and that if neither an exponent part nor a radix character appears in

47221 a decimal floating-point number, or if a binary exponent part does not appear in a hexadecimal
47222 floating-point number, an exponent part of the appropriate type with value zero is assumed to
47223 follow the last digit in the string. If the subject sequence begins with a minus sign, the sequence
47224 shall be interpreted as negated. A character sequence INF or INFINITY shall be interpreted as an
47225 infinity, if representable in the return type, else as if it were a floating constant that is too large
47226 for the range of the return type. A character sequence NAN or NAN(*n-char-sequence*$_{opt}$) shall be
47227 interpreted as a quiet NaN, if supported in the return type, else as if it were a subject sequence
47228 part that does not have the expected form; the meaning of the *n*-char sequences is
47229 implementation-defined. A pointer to the final string is stored in the object pointed to by *endptr*,
47230 provided that *endptr* is not a null pointer.

47231 If the subject sequence has the hexadecimal form and FLT_RADIX is a power of 2, the value
47232 resulting from the conversion is correctly rounded.

47233 CX The radix character is defined in the locale of the process (category *LC_NUMERIC*). In the
47234 POSIX locale, or in a locale where the radix character is not defined, the radix character shall
47235 default to a period (`'.'`).

47236 CX In other than the C or POSIX locales, other implementation-defined subject sequences may be
47237 accepted.

47238 If the subject sequence is empty or does not have the expected form, no conversion shall be
47239 performed; the value of *str* is stored in the object pointed to by *endptr*, provided that *endptr* is not
47240 a null pointer.

47241 CX The *strtod*( ) function shall not change the setting of *errno* if successful.

47242 Since 0 is returned on error and is also a valid return on success, an application wishing to check
47243 for error situations should set *errno* to 0, then call *strtod*( ), *strtof*( ), or *strtold*( ), then check *errno*.

47244 **RETURN VALUE**

47245 Upon successful completion, these functions shall return the converted value. If no conversion
47246 could be performed, 0 shall be returned, and *errno* may be set to [EINVAL].

47247 If the correct value is outside the range of representable values, ±HUGE_VAL, ±HUGE_VALF, or
47248 ±HUGE_VALL shall be returned (according to the sign of the value), and *errno* shall be set to
47249 [ERANGE].

47250 If the correct value would cause an underflow, a value whose magnitude is no greater than the
47251 smallest normalized positive number in the return type shall be returned and *errno* set to
47252 [ERANGE].

47253 **ERRORS**

47254 These functions shall fail if:

47255 CX [ERANGE] The value to be returned would cause overflow or underflow.

47256 These functions may fail if:

47257 CX [EINVAL] No conversion could be performed.

47258 **EXAMPLES**

47259 None.

47260 **APPLICATION USAGE**

47261 If the subject sequence has the hexadecimal form and FLT_RADIX is not a power of 2, and the
47262 result is not exactly representable, the result should be one of the two numbers in the
47263 appropriate internal format that are adjacent to the hexadecimal floating source value, with the
47264 extra stipulation that the error should have a correct sign for the current rounding direction.

47265 If the subject sequence has the decimal form and at most DECIMAL_DIG (defined in **<float.h>**)
47266 significant digits, the result should be correctly rounded. If the subject sequence *D* has the

47267 decimal form and more than DECIMAL_DIG significant digits, consider the two bounding,
47268 adjacent decimal strings *L* and *U*, both having DECIMAL_DIG significant digits, such that the
47269 values of *L*, *D*, and *U* satisfy *L* <= *D* <= *U*. The result should be one of the (equal or adjacent)
47270 values that would be obtained by correctly rounding *L* and *U* according to the current rounding
47271 direction, with the extra stipulation that the error with respect to *D* should have a correct sign
47272 for the current rounding direction.

47273 The changes to *strtod*( ) introduced by the ISO/IEC 9899: 1999 standard can alter the behavior of
47274 well-formed applications complying with the ISO/IEC 9899: 1990 standard and thus earlier
47275 versions of the base documents. One such example would be:

```
47276 int
47277 what_kind_of_number (char *s)
47278 {
47279     char *endp;
47280     double d;
47281     long l;

47282     d = strtod(s, &endp);
47283     if (s != endp && *endp == '\0')
47284         printf("It's a float with value %g\n", d);
47285     else
47286     {
47287         l = strtol(s, &endp, 0);
47288         if (s != endp && *endp == '\0')
47289             printf("It's an integer with value %ld\n", 1);
47290         else
47291             return 1;
47292     }
47293     return 0;
47294 }
```

47295 If the function is called with:

```
47296 what_kind_of_number ("0x10")
```

47297 an ISO/IEC 9899: 1990 standard-compliant library will result in the function printing:

```
47298 It's an integer with value 16
```

47299 With the ISO/IEC 9899: 1999 standard, the result is:

```
47300 It's a float with value 16
```

47301 The change in behavior is due to the inclusion of floating-point numbers in hexadecimal
47302 notation without requiring that either a decimal point or the binary exponent be present.

47303 **RATIONALE**

47304 None.

47305 **FUTURE DIRECTIONS**

47306 None.

47307 **SEE ALSO**

47308 *isspace*( ), *localeconv*( ), *scanf*( ), *setlocale*( ), *strtol*( ), the Base Definitions volume of
47309 IEEE Std 1003.1-200x, Chapter 7, Locale, **<float.h>**, **<stdlib.h>**

47310 **CHANGE HISTORY**

47311 First released in Issue 1. Derived from Issue 1 of the SVID.

47312 **Issue 5**

47313 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

47314 **Issue 6**

47315 Extensions beyond the ISO C standard are marked.

47316 The following new requirements on POSIX implementations derive from alignment with the
47317 Single UNIX Specification:

47318 • In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
47319 added if no conversion could be performed.

47320 The following changes are made for alignment with the ISO/IEC 9899: 1999 standard:

47321 • The *strtod*( ) function is updated.

47322 • The *strtof*( ) and *strtold*( ) functions are added.

47323 • The DESCRIPTION is extensively revised.

47324 ISO/IEC 9899: 1999 standard, Technical Corrigendum 1 is incorporated.

47325 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/61 is applied, correcting the second
47326 paragraph in the RETURN VALUE section. This change clarifies the sign of the return value.

47327 **Issue 7**

47328 Austin Group Interpretation 1003.1-2001 #015 is applied.

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

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 1003.1-200x defers to the ISO C standard.

These functions shall be equivalent to the *strtol*( ), *strtoll*( ), *strtoul*( ), and *strtoull*( ) functions, except that the initial portion of the string shall be converted to **intmax_t** and **uintmax_t** representation, respectively.

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 1003.1-200x, **<inttypes.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

NAME

strtok, strtok_r — split string into tokens

SYNOPSIS

```
#include <string.h>

char *strtok(char *restrict s1, const char *restrict s2);
CX char *strtok_r(char *restrict s, const char *restrict sep,
        char **restrict lasts);
```

DESCRIPTION

CX For *strtok*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x 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 NUL character, 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 1003.1-200x calls *strtok*( ).

CX The *strtok*( ) function need not be thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

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.

47409 **RETURN VALUE**

47410 Upon successful completion, *strtok*( ) shall return a pointer to the first byte of a token. Otherwise,
47411 if there is no token, *strtok*( ) shall return a null pointer.

47412 CX The *strtok_r*( ) function shall return a pointer to the token found, or a null pointer when no token
47413 is found.

47414 **ERRORS**

47415 No errors are defined.

47416 **EXAMPLES**

47417 **Searching for Word Separators**

47418 The following example searches for tokens separated by <space>s.

47419 ```
#include <string.h>
```
47420 ```
...
```
47421 ```
char *token;
```
47422 ```
char *line = "LINE TO BE SEPARATED";
```
47423 ```
char *search = " ";
```

47424 ```
/* Token will point to "LINE". */
```
47425 ```
token = strtok(line, search);
```

47426 ```
/* Token will point to "TO". */
```
47427 ```
token = strtok(NULL, search);
```

47428 **Breaking a Line**

47429 The following example uses *strtok*( ) to break a line into two character strings separated by any
47430 combination of <space>s, <tab>s, or <newline>s.

47431 ```
#include <string.h>
```
47432 ```
...
```
47433 ```
struct element {
```
47434 ```
    char *key;
```
47435 ```
    char *data;
```
47436 ```
};
```
47437 ```
...
```
47438 ```
char line[LINE_MAX];
```
47439 ```
char *key, *data;
```
47440 ```
...
```
47441 ```
key = strtok(line, " \n");
```
47442 ```
data = strtok(NULL, " \n");
```
47443 ```
...
```

47444 **APPLICATION USAGE**

47445 The *strtok_r*( ) function is thread-safe and stores its state in a user-supplied buffer instead of
47446 possibly using a static data area that may be overwritten by an unrelated call from another
47447 thread.

47448 **RATIONALE**

47449 The *strtok*( ) function searches for a separator string within a larger string. It returns a pointer to
47450 the last substring between separator strings. This function uses static storage to keep track of
47451 the current string position between calls. The new function, *strtok_r*( ), takes an additional
47452 argument, *lasts*, to keep track of the current position in the string.

47453 **FUTURE DIRECTIONS**

47454 None.

47455 **SEE ALSO**

47456 The Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

47457 **CHANGE HISTORY**

47458 First released in Issue 1. Derived from Issue 1 of the SVID.

47459 **Issue 5**

47460 The *strtok_r*( ) function is included for alignment with the POSIX Threads Extension.

47461 A note indicating that the *strtok*( ) function need not be reentrant is added to the DESCRIPTION.

47462 **Issue 6**

47463 Extensions beyond the ISO C standard are marked.

47464 The *strtok_r*( ) function is marked as part of the Thread-Safe Functions option.

47465 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

47466 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
47467 its avoidance of possibly using a static data area.

47468 The **restrict** keyword is added to the *strtok*( ) and *strtok_r*( ) prototypes for alignment with the
47469 ISO/IEC 9899: 1999 standard.

47470 **Issue 7**

47471 The *strtok_r*( ) function is moved from the Thread-Safe Functions option to the Base.

47472 **NAME**

47473 strtol, strtoll — convert a string to a long integer

47474 **SYNOPSIS**

47475
```
#include <stdlib.h>
```

47476
```
long strtol(const char *restrict str, char **restrict endptr, int base);
```
47477
```
long long strtoll(const char *restrict str, char **restrict endptr,
    int base)
```

47479 **DESCRIPTION**

47480 CX The functionality described on this reference page is aligned with the ISO C standard. Any
47481 conflict between the requirements described here and the ISO C standard is unintentional. This
47482 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

47483 These functions shall convert the initial portion of the string pointed to by *str* to a type **long** and
47484 **long long** representation, respectively. First, they decompose the input string into three parts:

47485 1. An initial, possibly empty, sequence of white-space characters (as specified by *isspace*( ))

47486 2. A subject sequence interpreted as an integer represented in some radix determined by the
47487 value of *base*

47488 3. A final string of one or more unrecognized characters, including the terminating NUL
47489 character of the input string.

47490 Then they shall attempt to convert the subject sequence to an integer, and return the result.

47491 If the value of *base* is 0, the expected form of the subject sequence is that of a decimal constant,
47492 octal constant, or hexadecimal constant, any of which may be preceded by a `'+'` or `'−'` sign. A
47493 decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An
47494 octal constant consists of the prefix `'0'` optionally followed by a sequence of the digits `'0'` to
47495 `'7'` only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the
47496 decimal digits and letters `'a'` (or `'A'`) to `'f'` (or `'F'`) with values 10 to 15 respectively.

47497 If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence
47498 of letters and digits representing an integer with the radix specified by *base*, optionally preceded
47499 by a `'+'` or `'−'` sign. The letters from `'a'` (or `'A'`) to `'z'` (or `'Z'`) inclusive are ascribed the
47500 values 10 to 35; only letters whose ascribed values are less than that of *base* are permitted. If the
47501 value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and
47502 digits, following the sign if present.

47503 The subject sequence is defined as the longest initial subsequence of the input string, starting
47504 with the first non-white-space character that is of the expected form. The subject sequence shall
47505 contain no characters if the input string is empty or consists entirely of white-space characters,
47506 or if the first non-white-space character is other than a sign or a permissible letter or digit.

47507 If the subject sequence has the expected form and the value of *base* is 0, the sequence of
47508 characters starting with the first digit shall be interpreted as an integer constant. If the subject
47509 sequence has the expected form and the value of *base* is between 2 and 36, it shall be used as the
47510 base for conversion, ascribing to each letter its value as given above. If the subject sequence
47511 begins with a minus sign, the value resulting from the conversion shall be negated. A pointer to
47512 the final string shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a null
47513 pointer.

47514 CX In other than the C or POSIX locales, other implementation-defined subject sequences may be
47515 accepted.

47516 If the subject sequence is empty or does not have the expected form, no conversion is performed;

the value of *str* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

CX 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 CX 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:

CX [EINVAL] The value of *base* is not supported.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalpha( ), *scanf*( ), *strtod*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

47557 **NAME**

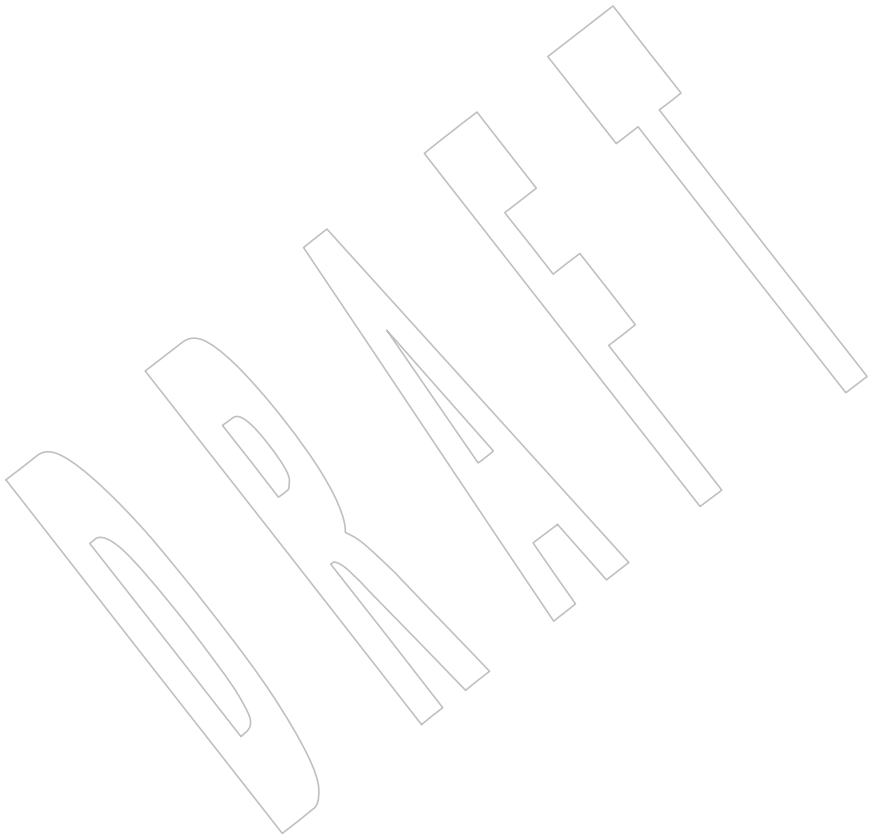
47558 strtold — convert a string to a double-precision number

47559 **SYNOPSIS**

47560 ```
#include <stdlib.h>
```

47561 ```
long double strtold(const char *restrict nptr, char **restrict endptr);
```

47562 **DESCRIPTION**

47563 Refer to *strtod( )*.



47564 **NAME**

47565 strtoll — convert a string to a long integer

47566 **SYNOPSIS**

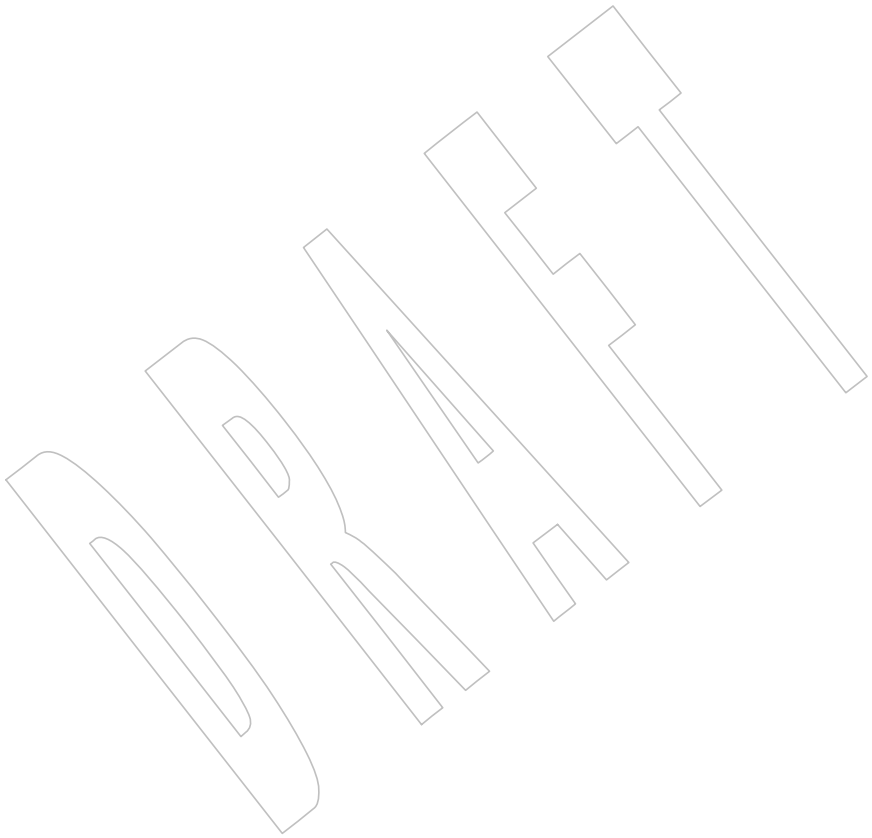
47567
```
#include <stdlib.h>
```

47568
```
long long strtoll(const char *restrict str, char **restrict endptr,
```

47569
```
    int base);
```

47570 **DESCRIPTION**

47571 Refer to *strtol( )*.



47572 **NAME**

47573 strtoul, strtoull — convert a string to an unsigned long

47574 **SYNOPSIS**

47575
```
#include <stdlib.h>
```

47576
```
unsigned long strtoul(const char *restrict str,
    char **restrict endptr, int base);
```
47577
47578
```
unsigned long long strtoull(const char *restrict str,
    char **restrict endptr, int base);
```
47579

47580 **DESCRIPTION**

47581 CX The functionality described on this reference page is aligned with the ISO C standard. Any
47582 conflict between the requirements described here and the ISO C standard is unintentional. This
47583 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

47584 These functions shall convert the initial portion of the string pointed to by *str* to a type **unsigned**
47585 **long** and **unsigned long long** representation, respectively. First, they decompose the input string
47586 into three parts:

47587 1. An initial, possibly empty, sequence of white-space characters (as specified by *isspace*( ))

47588 2. A subject sequence interpreted as an integer represented in some radix determined by the
47589 value of *base*

47590 3. A final string of one or more unrecognized characters, including the terminating NUL
47591 character of the input string

47592 Then they shall attempt to convert the subject sequence to an unsigned integer, and return the
47593 result.

47594 If the value of *base* is 0, the expected form of the subject sequence is that of a decimal constant,
47595 octal constant, or hexadecimal constant, any of which may be preceded by a `'+'` or `'−'` sign. A
47596 decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An
47597 octal constant consists of the prefix `'0'` optionally followed by a sequence of the digits `'0'` to
47598 `'7'` only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the
47599 decimal digits and letters `'a'` (or `'A'`) to `'f'` (or `'F'`) with values 10 to 15 respectively.

47600 If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence
47601 of letters and digits representing an integer with the radix specified by *base*, optionally preceded
47602 by a `'+'` or `'−'` sign. The letters from `'a'` (or `'A'`) to `'z'` (or `'Z'`) inclusive are ascribed the
47603 values 10 to 35; only letters whose ascribed values are less than that of *base* are permitted. If the
47604 value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and
47605 digits, following the sign if present.

47606 The subject sequence is defined as the longest initial subsequence of the input string, starting
47607 with the first non-white-space character that is of the expected form. The subject sequence shall
47608 contain no characters if the input string is empty or consists entirely of white-space characters,
47609 or if the first non-white-space character is other than a sign or a permissible letter or digit.

47610 If the subject sequence has the expected form and the value of *base* is 0, the sequence of
47611 characters starting with the first digit shall be interpreted as an integer constant. If the subject
47612 sequence has the expected form and the value of *base* is between 2 and 36, it shall be used as the
47613 base for conversion, ascribing to each letter its value as given above. If the subject sequence
47614 begins with a minus sign, the value resulting from the conversion shall be negated. A pointer to
47615 the final string shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a null
47616 pointer.

47617 CX In other than the C or POSIX locales, other implementation-defined subject sequences may be
47618 accepted.

47619 If the subject sequence is empty or does not have the expected form, no conversion shall be
47620 performed; the value of *str* shall be stored in the object pointed to by *endptr*, provided that *endptr*
47621 is not a null pointer.

47622 CX The *strtoul*( ) function shall not change the setting of *errno* if successful.

47623 Since 0, {ULONG_MAX}, and {ULLONG_MAX} are returned on error and are also valid returns
47624 on success, an application wishing to check for error situations should set *errno* to 0, then call
47625 *strtoul*( ) or *strtoull*( ), then check *errno*.

47626 **RETURN VALUE**

47627 Upon successful completion, these functions shall return the converted value, if any. If no
47628 CX conversion could be performed, 0 shall be returned and *errno* may be set to [EINVAL]. If the
47629 correct value is outside the range of representable values, {ULONG_MAX} or {ULLONG_MAX}
47630 shall be returned and *errno* set to [ERANGE].

47631 **ERRORS**

47632 These functions shall fail if:

47633 CX [EINVAL] The value of *base* is not supported.

47634 [ERANGE] The value to be returned is not representable.

47635 These functions may fail if:

47636 CX [EINVAL] No conversion could be performed.

47637 **EXAMPLES**

47638 None.

47639 **APPLICATION USAGE**

47640 None.

47641 **RATIONALE**

47642 None.

47643 **FUTURE DIRECTIONS**

47644 None.

47645 **SEE ALSO**

47646 *isalpha*( ), *scanf*( ), *strtod*( ), *strtol*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
47647 **<stdlib.h>**

47648 **CHANGE HISTORY**

47649 First released in Issue 4. Derived from the ANSI C standard.

47650 **Issue 5**

47651 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

47652 **Issue 6**

47653 Extensions beyond the ISO C standard are marked.

47654 The following new requirements on POSIX implementations derive from alignment with the
47655 Single UNIX Specification:

47656 - The [EINVAL] error condition is added for when the value of *base* is not supported.

47657 In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
47658 added if no conversion could be performed.

47659 The following changes are made for alignment with the ISO/IEC 9899: 1999 standard:

- The *strtoul*( ) prototype is updated.
- The *strtoull*( ) function is added.

47662 **NAME**

47663 strtoumax — convert a string to an integer type

47664 **SYNOPSIS**

47665
```
#include <inttypes.h>
```

47666
```
uintmax_t strtoumax(const char *restrict nptr, char **restrict endptr,
    int base);
```

47667

47668 **DESCRIPTION**

47669 Refer to *strtoimax( )*.

NAME

strxfrm, strxfrm_l — string transformation

SYNOPSIS

```
#include <string.h>

size_t strxfrm(char *restrict s1, const char *restrict s2, size_t n);
CX size_t strxfrm_l(char *restrict s1, const char *restrict s2,
        size_t n, locale_t locale);
```

DESCRIPTION

CX For *strxfrm*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

CX The *strxfrm*( ) and *strxfrm_l*( ) functions 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, CX corresponding to the result of *strcoll*( ) or *strcoll_l*( ), respectively, applied to the same two CX original strings with the same locale. No more than *n* bytes are placed into the resulting array pointed to by *s1*, including the terminating NUL character. If *n* is 0, *s1* is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

CX The *strxfrm*( ) and *strxfrm_l*( ) functions 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 CX situations should set *errno* to 0, then call *strxfrm*( ) or *strxfrm_l*( ), then check *errno*.

RETURN VALUE

CX Upon successful completion, *strxfrm*( ) and *strxfrm_l*( ) shall return the length of the transformed string (not including the terminating NUL character). If the value returned is *n* or more, the contents of the array pointed to by *s1* are unspecified.

CX On error, *strxfrm*( ) and *strxfrm_l*( ) may set *errno* but no return value is reserved to indicate an error.

ERRORS

These functions may fail if:

CX [EINVAL] The string pointed to by the *s2* argument contains characters outside the domain of the collating sequence.

The *strxfrm_l*( ) function may fail if:

CX [EINVAL] *locale* is not a valid locale object.

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 locale of the process (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.

47711 **RATIONALE**

47712 None.

47713 **FUTURE DIRECTIONS**

47714 None.

47715 **SEE ALSO**

47716 *strcmp( )*, *strcoll( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

47717 **CHANGE HISTORY**

47718 First released in Issue 3. Included for alignment with the ISO C standard.

47719 **Issue 5**

47720 The DESCRIPTION is updated to indicate that *errno* does not change if the function is successful.

47721 **Issue 6**

47722 Extensions beyond the ISO C standard are marked.

47723 The following new requirements on POSIX implementations derive from alignment with the
47724 Single UNIX Specification:

47725 • In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
47726 added if no conversion could be performed.

47727 The *strxfrm( )* prototype is updated for alignment with the ISO/IEC 9899: 1999 standard.

47728 **Issue 7**

47729 The *strxfrm_l( )* function is added from The Open Group Technical Standard, 2006, Extended API
47730 Set Part 4.

47731 **NAME**

47732 swab — swap bytes

47733 **SYNOPSIS**

47734 XSI
```
#include <unistd.h>

void swab(const void *restrict src, void *restrict dest,
    ssize_t nbytes);
```

47737 **DESCRIPTION**

47738 The *swab*( ) function shall copy *nbytes* bytes, which are pointed to by *src*, to the object pointed to
47739 by *dest*, exchanging adjacent bytes. The *nbytes* argument should be even. If *nbytes* is odd, *swab*( )
47740 copies and exchanges *nbytes*–1 bytes and the disposition of the last byte is unspecified. If
47741 copying takes place between objects that overlap, the behavior is undefined. If *nbytes* is
47742 negative, *swab*( ) does nothing.

47743 **RETURN VALUE**

47744 None.

47745 **ERRORS**

47746 No errors are defined.

47747 **EXAMPLES**

47748 None.

47749 **APPLICATION USAGE**

47750 None.

47751 **RATIONALE**

47752 None.

47753 **FUTURE DIRECTIONS**

47754 None.

47755 **SEE ALSO**

47756 The Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

47757 **CHANGE HISTORY**

47758 First released in Issue 1. Derived from Issue 1 of the SVID.

47759 **Issue 6**

47760 The **restrict** keyword is added to the *swab*( ) prototype for alignment with the
47761 ISO/IEC 9899: 1999 standard.

47762 **NAME**

47763 swprintf — print formatted wide-character output

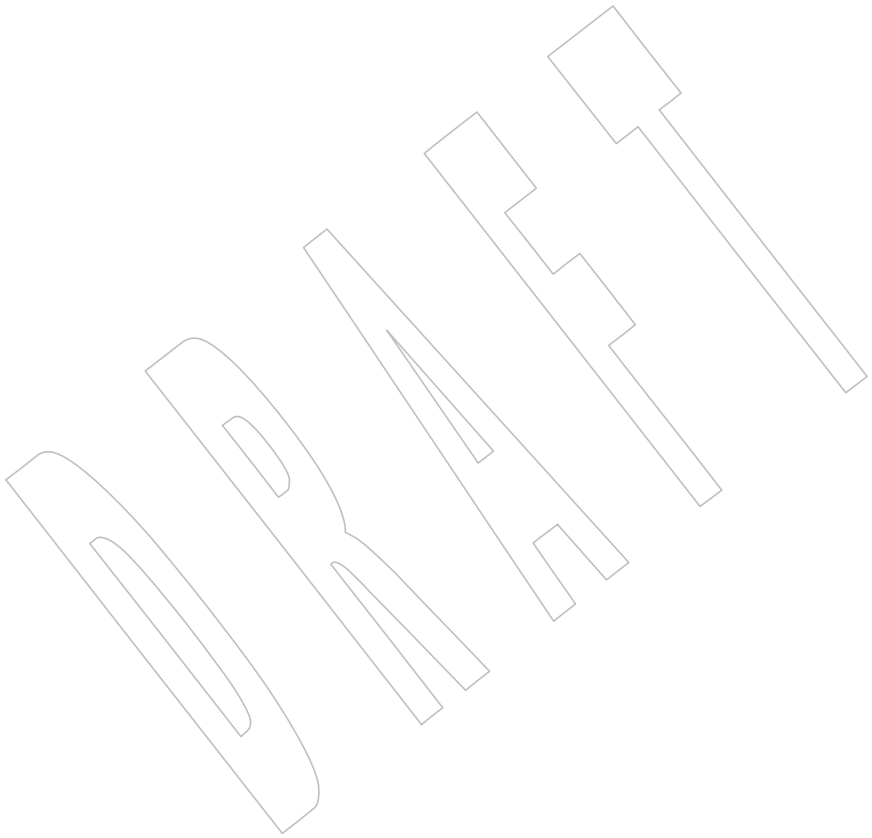
47764 **SYNOPSIS**

47765 ```
#include <stdio.h>
```

47766 ```
#include <wchar.h>
```

47767 ```
int swprintf(wchar_t *restrict ws, size_t n,
```

47768 ```
    const wchar_t *restrict format, ...);
```

47769 **DESCRIPTION**

47770 Refer to *fwprintf( )*.



47771 **NAME**

47772 swscanf — convert formatted wide-character input

47773 **SYNOPSIS**

47774 ```
#include <stdio.h>
```

47775 ```
#include <wchar.h>
```

47776 ```
int swscanf(const wchar_t *restrict ws,
```

47777 ```
    const wchar_t *restrict format, ... );
```

47778 **DESCRIPTION**

47779 Refer to *fwscanf( )*.

47780 **NAME**

47781 symlink, symlinkat — make a symbolic link relative to directory file descriptor

47782 **SYNOPSIS**

47783
```
#include <unistd.h>

int symlink(const char *path1, const char *path2);
int symlinkat(const char *path1, int fd, const char *path2);
```

47786 **DESCRIPTION**

47787 The *symlink*( ) function shall create a symbolic link called *path2* that contains the string pointed to
47788 by *path1* (*path2* is the name of the symbolic link created, *path1* is the string contained in the
47789 symbolic link).

47790 The string pointed to by *path1* shall be treated only as a character string and shall not be
47791 validated as a pathname.

47792 If the *symlink*( ) function fails for any reason other than [EIO], any file named by *path2* shall be
47793 unaffected.

47794 The *symlinkat*( ) function shall be equivalent to the *symlink*( ) function except in the case where
47795 *path2* specifies a relative path. In this case the symbolic link is created relative to the directory
47796 associated with the file descriptor *fd* instead of the current working directory. It is unspecified
47797 whether directory searches are permitted based on whether the file was opened with search
47798 permission or on the current permissions of the directory underlying the file descriptor.

47799 If *symlinkat*( ) is passed the special value AT_FDCWD in the *fd* parameter, the current working
47800 directory is used and the behavior shall be identical to a call to *symlink*( ).

47801 **RETURN VALUE**

47802 Upon successful completion, these functions shall return 0. Otherwise, these functions shall
47803 return –1 and set *errno* to indicate the error.

47804 **ERRORS**

47805 These functions shall fail if:

47806 [EACCES] Write permission is denied in the directory where the symbolic link is being
47807 created, or search permission is denied for a component of the path prefix of
47808 *path2*.

47809 [EEXIST] The *path2* argument names an existing file or symbolic link.

47810 [EIO] An I/O error occurs while reading from or writing to the file system.

47811 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path2*
47812 argument.

47813 [ENAMETOOLONG]
47814 The length of the *path2* argument exceeds {PATH_MAX} or a pathname
47815 component is longer than {NAME_MAX} or the length of the *path1* argument
47816 is longer than {SYMLINK_MAX}.

47817 [ENOENT] A component of *path2* does not name an existing file or *path2* is an empty
47818 string.

47819 [ENOSPC] The directory in which the entry for the new symbolic link is being placed
47820 cannot be extended because no space is left on the file system containing the
47821 directory, or the new symbolic link cannot be created because no space is left
47822 on the file system which shall contain the link, or the file system is out of file-
47823 allocation resources.

47824 [ENOTDIR] A component of the path prefix of *path2* is not a directory.

47825 [EROFS] The new symbolic link would reside on a read-only file system.

47826 The *symlinkat*( ) function shall fail if:

47827 [EBADF] The *path2* argument does not specify an absolute path and the *fd* argument is
47828 neither AT_FDCWD nor a valid file descriptor open for searching.

47829 These functions may fail if:

47830 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
47831 resolution of the *path2* argument.

47832 [ENAMETOOLONG]
47833 As a result of encountering a symbolic link in resolution of the *path2*
47834 argument, the length of the substituted pathname string exceeded
47835 {PATH_MAX} bytes (including the terminating null byte), or the length of the
47836 string pointed to by *path1* exceeded {SYMLINK_MAX}.

47837 The *symlinkat*( ) function may fail if:

47838 [ENOTDIR] The *path2* argument is not an absolute path and *fd* is neither AT_FDCWD nor a
47839 file descriptor associated with a directory.

47840 **EXAMPLES**

47841 None.

47842 **APPLICATION USAGE**

47843 Like a hard link, a symbolic link allows a file to have multiple logical names. The presence of a
47844 hard link guarantees the existence of a file, even after the original name has been removed. A
47845 symbolic link provides no such assurance; in fact, the file named by the *path1* argument need not
47846 exist when the link is created. A symbolic link can cross file system boundaries.

47847 Normal permission checks are made on each component of the symbolic link pathname during
47848 its resolution.

47849 **RATIONALE**

47850 Since IEEE Std 1003.1-200x does not require any association of file times with symbolic links,
47851 there is no requirement that file times be updated by *symlink*( ).

47852 The purpose of the *symlinkat*( ) function is to create symbolic links in directories other than the
47853 current working directory without exposure to race conditions. Any part of the path of a file
47854 could be changed in parallel to a call to *symlink*( ), resulting in unspecified behavior. By opening
47855 a file descriptor for the target directory and using the *symlinkat*( ) function it can be guaranteed
47856 that the created symbolic link is located relative to the desired directory.

47857 **FUTURE DIRECTIONS**

47858 None.

47859 **SEE ALSO**

47860 *fdopendir*( ), *fstatat*( ), *lchown*( ), *link*( ), *open*( ), *readlink*( ), *rename*( ), *unlink*( ), the Base Definitions
47861 volume of IEEE Std 1003.1-200x, **<unistd.h>**

47862 **CHANGE HISTORY**

47863 First released in Issue 4, Version 2.

47864 **Issue 5**

47865 Moved from X/OPEN UNIX extension to BASE.

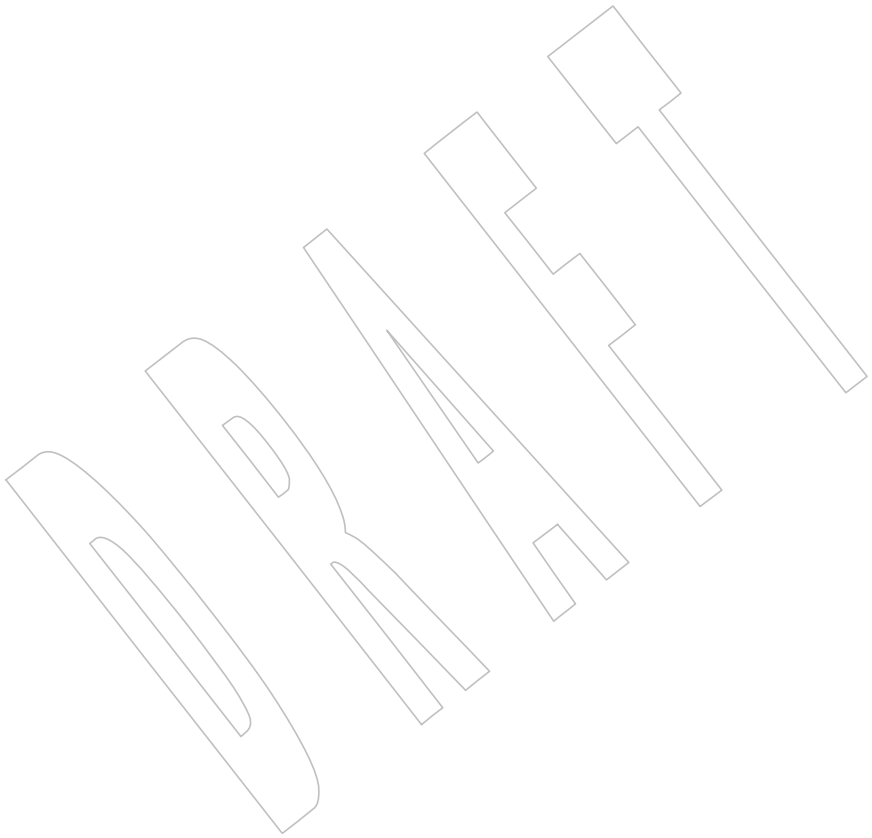
Issue 6

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.

Issue 7

The *symlinkat*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 2.



47873 **NAME**

47874 symlinkat — make a symbolic link relative to directory file descriptor

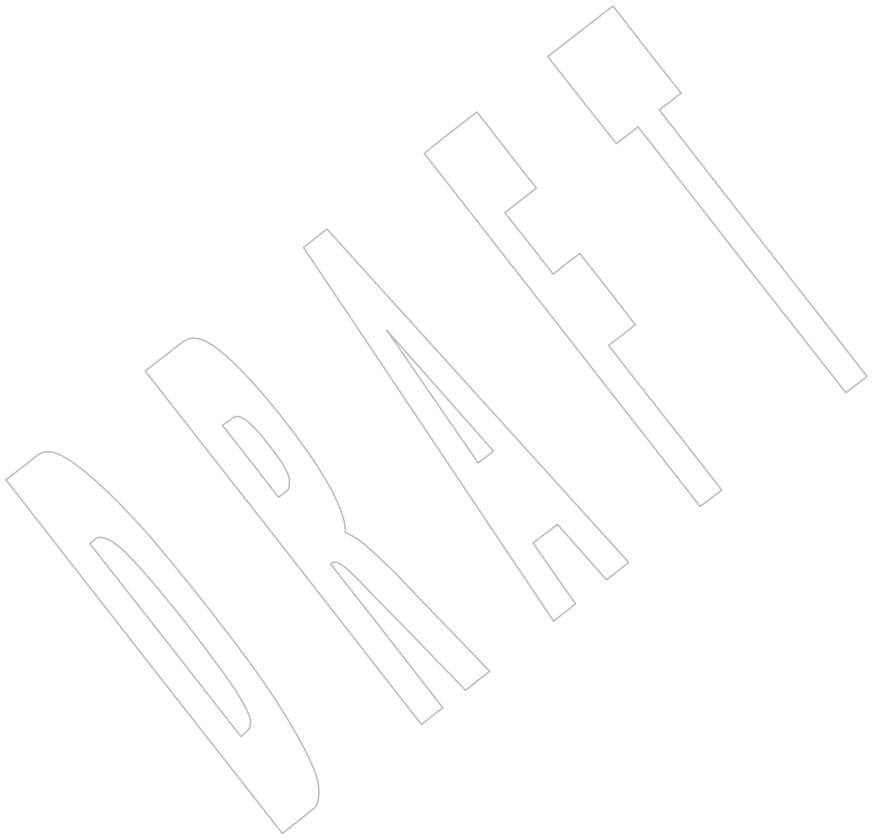
47875 **SYNOPSIS**

47876
```
#include <unistd.h>
```

47877
```
int symlinkat(const char *path1, int fd, const char *path2);
```

47878 **DESCRIPTION**

47879 Refer to *symlink( )*.



47880 **NAME**

47881 sync — schedule file system updates

47882 **SYNOPSIS**

47883 XSI
```
#include <unistd.h>

void sync(void);
```

47884

47885 **DESCRIPTION**

47886 The *sync*( ) function shall cause all information in memory that updates file systems to be
47887 scheduled for writing out to all file systems.

47888 The writing, although scheduled, is not necessarily complete upon return from *sync*( ).

47889 **RETURN VALUE**

47890 The *sync*( ) function shall not return a value.

47891 **ERRORS**

47892 No errors are defined.

47893 **EXAMPLES**

47894 None.

47895 **APPLICATION USAGE**

47896 None.

47897 **RATIONALE**

47898 None.

47899 **FUTURE DIRECTIONS**

47900 None.

47901 **SEE ALSO**

47902 *fsync*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

47903 **CHANGE HISTORY**

47904 First released in Issue 4, Version 2.

47905 **Issue 5**

47906 Moved from X/OPEN UNIX extension to BASE.

47907 **NAME**

47908 sysconf — get configurable system variables

47909 **SYNOPSIS**

47910
```
#include <unistd.h>
```

47911
```
long sysconf(int name);
```

47912 **DESCRIPTION**

47913 The *sysconf*( ) function provides a method for the application to determine the current value of a
47914 configurable system limit or option (*variable*). The implementation shall support all of the
47915 variables listed in the following table and may support others.

47916 The *name* argument represents the system variable to be queried. The following table lists the
47917 minimal set of system variables from **<limits.h>** or **<unistd.h>** that can be returned by *sysconf*( ),
47918 and the symbolic constants defined in **<unistd.h>** that are the corresponding values used for
47919 *name*.

47920

Variable	Value of Name
47921 {AIO_LISTIO_MAX}	_SC_AIO_LISTIO_MAX
47922 {AIO_MAX}	_SC_AIO_MAX
47923 {AIO_PRIO_DELTA_MAX}	_SC_AIO_PRIO_DELTA_MAX
47924 {ARG_MAX}	_SC_ARG_MAX
47925 {ATEXIT_MAX}	_SC_ATEXIT_MAX
47926 {BC_BASE_MAX}	_SC_BC_BASE_MAX
47927 {BC_DIM_MAX}	_SC_BC_DIM_MAX
47928 {BC_SCALE_MAX}	_SC_BC_SCALE_MAX
47929 {BC_STRING_MAX}	_SC_BC_STRING_MAX
47930 {CHILD_MAX}	_SC_CHILD_MAX
47931 Clock ticks/second	_SC_CLK_TCK
47932 {COLL_WEIGHTS_MAX}	_SC_COLL_WEIGHTS_MAX
47933 {DELAYTIMER_MAX}	_SC_DELAYTIMER_MAX
47934 {EXPR_NEST_MAX}	_SC_EXPR_NEST_MAX
47935 {HOST_NAME_MAX}	_SC_HOST_NAME_MAX
47936 {IOV_MAX}	_SC_IOV_MAX
47937 {LINE_MAX}	_SC_LINE_MAX
47938 {LOGIN_NAME_MAX}	_SC_LOGIN_NAME_MAX
47939 {NGROUPS_MAX}	_SC_NGROUPS_MAX
47940 Maximum size of *getgrgid_r*( ) and 47941 *getgrnam_r*( ) data buffers	_SC_GETGR_R_SIZE_MAX
47942 Maximum size of *getpwuid_r*( ) and 47943 *getpwnam_r*( ) data buffers	_SC_GETPW_R_SIZE_MAX
47944 {MQ_OPEN_MAX}	_SC_MQ_OPEN_MAX
47945 {MQ_PRIO_MAX}	_SC_MQ_PRIO_MAX
47946 {OPEN_MAX}	_SC_OPEN_MAX
47947 _POSIX_ADVISORY_INFO	_SC_ADVISORY_INFO
47948 _POSIX_BARRIERS	_SC_BARRIERS
47949 _POSIX_ASYNCHRONOUS_IO	_SC_ASYNCHRONOUS_IO
47950 _POSIX_CLOCK_SELECTION	_SC_CLOCK_SELECTION
47951 _POSIX_CPUTIME	_SC_CPUTIME
47952 _POSIX_FSYNC	_SC_FSYNC
47953 _POSIX_IPV6	_SC_IPV6
47954 _POSIX_JOB_CONTROL	_SC_JOB_CONTROL

Variable	Value of Name
_POSIX_MAPPED_FILES	_SC_MAPPED_FILES
_POSIX_MEMLOCK	_SC_MEMLOCK
_POSIX_MEMLOCK_RANGE	_SC_MEMLOCK_RANGE
_POSIX_MEMORY_PROTECTION	_SC_MEMORY_PROTECTION
_POSIX_MESSAGE_PASSING	_SC_MESSAGE_PASSING
_POSIX_MONOTONIC_CLOCK	_SC_MONOTONIC_CLOCK
_POSIX_PRIORITIZED_IO	_SC_PRIORITIZED_IO
_POSIX_PRIORITY_SCHEDULING	_SC_PRIORITY_SCHEDULING
_POSIX_RAW_SOCKETS	_SC_RAW_SOCKETS
_POSIX_READER_WRITER_LOCKS	_SC_READER_WRITER_LOCKS
_POSIX_REALTIME_SIGNALS	_SC_REALTIME_SIGNALS
_POSIX_REGEXP	_SC_REGEXP
_POSIX_SAVED_IDS	_SC_SAVED_IDS
_POSIX_SEMAPHORES	_SC_SEMAPHORES
_POSIX_SHARED_MEMORY_OBJECTS	_SC_SHARED_MEMORY_OBJECTS
_POSIX_SHELL	_SC_SHELL
_POSIX_SPAWN	_SC_SPAWN
_POSIX_SPIN_LOCKS	_SC_SPIN_LOCKS
_POSIX_SPORADIC_SERVER	_SC_SPORADIC_SERVER
_POSIX_SS_REPL_MAX	_SC_SS_REPL_MAX
_POSIX_SYNCHRONIZED_IO	_SC_SYNCHRONIZED_IO
_POSIX_THREAD_ATTR_STACKADDR	_SC_THREAD_ATTR_STACKADDR
_POSIX_THREAD_ATTR_STACKSIZE	_SC_THREAD_ATTR_STACKSIZE
_POSIX_THREAD_CPUTIME	_SC_THREAD_CPUTIME
_POSIX_THREAD_PRIO_INHERIT	_SC_THREAD_PRIO_INHERIT
_POSIX_THREAD_PRIO_PROTECT	_SC_THREAD_PRIO_PROTECT
_POSIX_THREAD_PRIORITY_SCHEDULING	_SC_THREAD_PRIORITY_SCHEDULING
_POSIX_THREAD_PROCESS_SHARED	_SC_THREAD_PROCESS_SHARED
_POSIX_THREAD_ROBUST_PRIO_INHERIT	_SC_THREAD_ROBUST_PRIO_INHERIT
_POSIX_THREAD_ROBUST_PRIO_PROTECT	_SC_THREAD_ROBUST_PRIO_PROTECT
_POSIX_THREAD_SAFE_FUNCTIONS	_SC_THREAD_SAFE_FUNCTIONS
_POSIX_THREAD_SPORADIC_SERVER	_SC_THREAD_SPORADIC_SERVER
_POSIX_THREADS	_SC_THREADS
_POSIX_TIMEOUTS	_SC_TIMEOUTS
_POSIX_TIMERS	_SC_TIMERS
_POSIX_TRACE	_SC_TRACE
_POSIX_TRACE_EVENT_FILTER	_SC_TRACE_EVENT_FILTER
_POSIX_TRACE_EVENT_NAME_MAX	_SC_TRACE_EVENT_NAME_MAX
_POSIX_TRACE_INHERIT	_SC_TRACE_INHERIT
_POSIX_TRACE_LOG	_SC_TRACE_LOG
_POSIX_TRACE_NAME_MAX	_SC_TRACE_NAME_MAX
_POSIX_TRACE_SYS_MAX	_SC_TRACE_SYS_MAX
_POSIX_TRACE_USER_EVENT_MAX	_SC_TRACE_USER_EVENT_MAX
_POSIX_TYPED_MEMORY_OBJECTS	_SC_TYPED_MEMORY_OBJECTS
_POSIX_VERSION	_SC_VERSION
_POSIX_V7_ILP32_OFF32	_SC_V7_ILP32_OFF32
_POSIX_V7_ILP32_OFFBIG	_SC_V7_ILP32_OFFBIG
_POSIX_V7_LP64_OFF64	_SC_V7_LP64_OFF64
_POSIX_V7_LPBIG_OFFBIG	_SC_V7_LPBIG_OFFBIG

	Variable	Value of Name
OB	_POSIX_V6_ILP32_OFF32	_SC_V6_ILP32_OFF32
	_POSIX_V6_ILP32_OFFBIG	_SC_V6_ILP32_OFFBIG
	_POSIX_V6_LP64_OFF64	_SC_V6_LP64_OFF64
	_POSIX_V6_LPBIG_OFFBIG	_SC_V6_LPBIG_OFFBIG
	_POSIX2_C_BIND	_SC_2_C_BIND
	_POSIX2_C_DEV	_SC_2_C_DEV
	_POSIX2_CHAR_TERM	_SC_2_CHAR_TERM
	_POSIX2_FORT_DEV	_SC_2_FORT_DEV
	_POSIX2_FORT_RUN	_SC_2_FORT_RUN
	_POSIX2_LOCALEDEF	_SC_2_LOCALEDEF
	_POSIX2_PBS	_SC_2_PBS
	_POSIX2_PBS_ACCOUNTING	_SC_2_PBS_ACCOUNTING
	_POSIX2_PBS_CHECKPOINT	_SC_2_PBS_CHECKPOINT
	_POSIX2_PBS_LOCATE	_SC_2_PBS_LOCATE
	_POSIX2_PBS_MESSAGE	_SC_2_PBS_MESSAGE
	_POSIX2_PBS_TRACK	_SC_2_PBS_TRACK
	_POSIX2_SW_DEV	_SC_2_SW_DEV
	_POSIX2_UPE	_SC_2_UPE
	_POSIX2_VERSION	_SC_2_VERSION
	{PAGE_SIZE}	_SC_PAGE_SIZE
	{PAGESIZE}	_SC_PAGESIZE
	{PTHREAD_DESTRUCTOR_ITERATIONS}	_SC_THREAD_DESTRUCTOR_ITERATIONS
	{PTHREAD_KEYS_MAX}	_SC_THREAD_KEYS_MAX
	{PTHREAD_STACK_MIN}	_SC_THREAD_STACK_MIN
	{PTHREAD_THREADS_MAX}	_SC_THREAD_THREADS_MAX
	{RE_DUP_MAX}	_SC_RE_DUP_MAX
	{RTSIG_MAX}	_SC_RTSIG_MAX
	{SEM_NSEMS_MAX}	_SC_SEM_NSEMS_MAX
	{SEM_VALUE_MAX}	_SC_SEM_VALUE_MAX
	{SIGQUEUE_MAX}	_SC_SIGQUEUE_MAX
	{STREAM_MAX}	_SC_STREAM_MAX
	{SYMLOOP_MAX}	_SC_SYMLOOP_MAX
	{TIMER_MAX}	_SC_TIMER_MAX
	{TTY_NAME_MAX}	_SC_TTY_NAME_MAX
	{TZNAME_MAX}	_SC_TZNAME_MAX
	_XOPEN_CRYPT	_SC_XOPEN_CRYPT
	_XOPEN_ENH_I18N	_SC_XOPEN_ENH_I18N
	_XOPEN_REALTIME	_SC_XOPEN_REALTIME
	_XOPEN_REALTIME_THREADS	_SC_XOPEN_REALTIME_THREADS
	_XOPEN_SHM	_SC_XOPEN_SHM
	_XOPEN_STREAMS	_SC_XOPEN_STREAMS
	_XOPEN_UNIX	_SC_XOPEN_UNIX
	_XOPEN_VERSION	_SC_XOPEN_VERSION

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 XSI during the lifetime of the calling process, except that *sysconf*(_SC_OPEN_MAX) may return different values before and after a call to *setrlimit*( ) which changes the RLIMIT_NOFILE soft

limit.

If the variable corresponding to *name* is dependent on an unsupported option, the results are unspecified.

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.

Application writers should check whether an option, such as _POSIX_TRACE, is supported prior to obtaining and using values for related variables, such as _POSIX_TRACE_NAME_MAX.

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 1003.1-200x, 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 1003.1-200x.

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 1003.1-200x 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 1003.1-200x, **<limits.h>**, **<unistd.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x, *getconf*

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 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.

Issue 6

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_REGEXP, _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.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/62 is applied, updating the DESCRIPTION to denote that the _PC* and _SC* symbols are now required to be supported. A corresponding change has been made in the Base Definitions volume of IEEE Std 1003.1-200x. The deletion in the second paragraph removes some duplicated text. Additional symbols that were erroneously omitted from this reference page have been added.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/63 is applied, making it clear in the RETURN VALUE section that the value returned for *sysconf*(_SC_OPEN_MAX) may change if a call to *setrlimit*( ) adjusts the RLIMIT_NOFILE soft limit.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/134 is applied, updating the DESCRIPTION to remove an erroneous entry for _POSIX_SYMLOOP_MAX. This corrects an error in IEEE Std 1003.1-2001/Cor 1-2002.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/135 is applied, removing _POSIX_FILE_LOCKING, _POSIX_MULTI_PROCESS, _POSIX2_C_VERSION, and _XOPEN_XCU_VERSION (and their associated _SC_* variables) from the DESCRIPTION and APPLICATION USAGE sections.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/136 is applied, adding the following constants (and their associated _SC_* variables) to the DESCRIPTION:

_POSIX_SS_REPL_MAX
_POSIX_TRACE_EVENT_NAME_MAX
_POSIX_TRACE_NAME_MAX
_POSIX_TRACE_SYS_MAX
_POSIX_TRACE_USER_EVENT_MAX

The RETURN VALUE and APPLICATION USAGE sections are updated to note that if variables are dependent on unsupported options, the results are unspecified.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/137 is applied, removing _REGEX_VERSION and _SC_REGEX_VERSION.

Issue 7

The variables for the supported programming environments are updated to be V7 and the LEGACY variables are removed.

The following constants are added:

_POSIX_THREAD_ROBUST_PRIO_INHERIT
_POSIX_THREAD_ROBUST_PRIO_PROTECT

48205 **NAME**

48206 syslog — log a message

48207 **SYNOPSIS**

48208 XSI
```
#include <syslog.h>

void syslog(int priority, const char *message, ... /* argument */);
```

48209

48210 **DESCRIPTION**

48211 Refer to *closelog( )*.

NAME

system — issue a command

SYNOPSIS

```
#include <stdlib.h>

int system(const char *command);
```

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 1003.1-200x defers to the ISO C standard.

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.

CX The *system*( ) function shall behave 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. It is unspecified whether the handlers registered with *pthread_atfork*( ) are called as part of the creation of the child process.

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.

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

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. CX The *system*( ) function shall always return non-zero when *command* is NULL.

CX 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.

48253 **ERRORS**

48254 CX The *system*( ) function may set *errno* values as described by *fork*( ).

48255 In addition, *system*( ) may fail if:

48256 CX [ECHILD] The status of the child process created by *system*( ) is no longer available.

48257 **EXAMPLES**

48258 None.

48259 **APPLICATION USAGE**

48260 If the return value of *system*( ) is not –1, its value can be decoded through the use of the macros
48261 described in **<sys/wait.h>**. For convenience, these macros are also provided in **<stdlib.h>**.

48262 Note that, while *system*( ) must ignore SIGINT and SIGQUIT and block SIGCHLD while waiting
48263 for the child to terminate, the handling of signals in the executed command is as specified by
48264 *fork*( ) and *exec*. For example, if SIGINT is being caught or is set to SIG_DFL when *system*( ) is
48265 called, then the child is started with SIGINT handling set to SIG_DFL.

48266 Ignoring SIGINT and SIGQUIT in the parent process prevents coordination problems (two
48267 processes reading from the same terminal, for example) when the executed command ignores or
48268 catches one of the signals. It is also usually the correct action when the user has given a
48269 command to the application to be executed synchronously (as in the `'!'` command in many
48270 interactive applications). In either case, the signal should be delivered only to the child process,
48271 not to the application itself. There is one situation where ignoring the signals might have less
48272 than the desired effect. This is when the application uses *system*( ) to perform some task invisible
48273 to the user. If the user typed the interrupt character (`"^C"`, for example) while *system*( ) is being
48274 used in this way, one would expect the application to be killed, but only the executed command
48275 is killed. Applications that use *system*( ) in this way should carefully check the return status from
48276 *system*( ) to see if the executed command was successful, and should take appropriate action
48277 when the command fails.

48278 Blocking SIGCHLD while waiting for the child to terminate prevents the application from
48279 catching the signal and obtaining status from *system*( )'s child process before *system*( ) can get the
48280 status itself.

48281 The context in which the utility is ultimately executed may differ from that in which *system*( )
48282 was called. For example, file descriptors that have the FD_CLOEXEC flag set are closed, and the
48283 process ID and parent process ID are different. Also, if the executed utility changes its
48284 environment variables or its current working directory, that change is not reflected in the caller's
48285 context.

48286 There is no defined way for an application to find the specific path for the shell. However,
48287 *confstr*( ) can provide a value for *PATH* that is guaranteed to find the *sh* utility.

48288 Using the *system*( ) function in more than one thread in a process or when the SIGCHLD signal is
48289 being manipulated by more than one thread in a process may produce unexpected results.

48290 **RATIONALE**

48291 The *system*( ) function should not be used by programs that have set user (or group) ID
48292 privileges. The *fork*( ) and *exec* family of functions (except *execlp*( ) and *execvp*( )), should be used
48293 instead. This prevents any unforeseen manipulation of the environment of the user that could
48294 cause execution of commands not anticipated by the calling program.

48295 There are three levels of specification for the *system*( ) function. The ISO C standard gives the
48296 most basic. It requires that the function exists, and defines a way for an application to query
48297 whether a command language interpreter exists. It says nothing about the command language
48298 or the environment in which the command is interpreted.

48299 IEEE Std 1003.1-200x places additional restrictions on *system*( ). It requires that if there is a
48300 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 1003.1-200x requires the command to be interpreted as in the shell command language defined in the Shell and Utilities volume of IEEE Std 1003.1-200x.

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 1003.1-200x 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 1003.1-200x, *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 1003.1-200x.

```
#include <signal.h>
int system(const char *cmd)
{
    int stat;
    pid_t pid;
    struct sigaction sa, savintr, savequit;
    sigset_t saveblock;
    if (cmd == NULL)
        return(1);
    sa.sa_handler = SIG_IGN;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigemptyset(&savintr.sa_mask);
```

```
        sigemptyset(&savequit.sa_mask);
        sigaction(SIGINT, &sa, &savintr);
        sigaction(SIGQUIT, &sa, &savequit);
        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 1003.1-200x, 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 1003.1-200x 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*( ), *pthread_atfork*( ), *waitpid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<limits.h>**, **<signal.h>**, **<stdlib.h>**, **<sys/wait.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x, *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.

Issue 7

SD5-XSH-ERN-30 is applied.

Austin Group Interpretation 1003.1-2001 #055 is applied, clarifying the thread-safety of this function and treatment of *at_fork*( ) handlers.

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

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 1003.1-200x defers to the ISO C standard.

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*(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 tangent of *x*.

If the correct value would cause underflow, and is not representable, a range error may occur, and MX either 0.0 (if supported), or an implementation-defined value shall be returned.

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.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value shall be returned.

XSI If the correct value would cause overflow, a range error shall occur and *tan*( ), *tanf*( ), and *tanl*( ) shall return ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL, respectively, with the same sign as the correct value of the function.

ERRORS

These functions shall fail if:

MX

Domain Error	The value of *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.

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.

These functions may fail if:

MX Range Error The result underflows, or 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

Taking the Tangent of a 45-Degree Angle

```
#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

atan( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, 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 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.

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

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/64 is applied, correcting the last paragraph in the RETURN VALUE section.

NAME

tanh, tanhf, tanhl — hyperbolic tangent functions

SYNOPSIS

```
#include <math.h>

double tanh(double x);
float tanhf(float x);
long double tanhl(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 1003.1-200x 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*.

MX 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:

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 (*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 1003.1-200x, 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.

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

48541 **NAME**

48542 tanl — tangent function

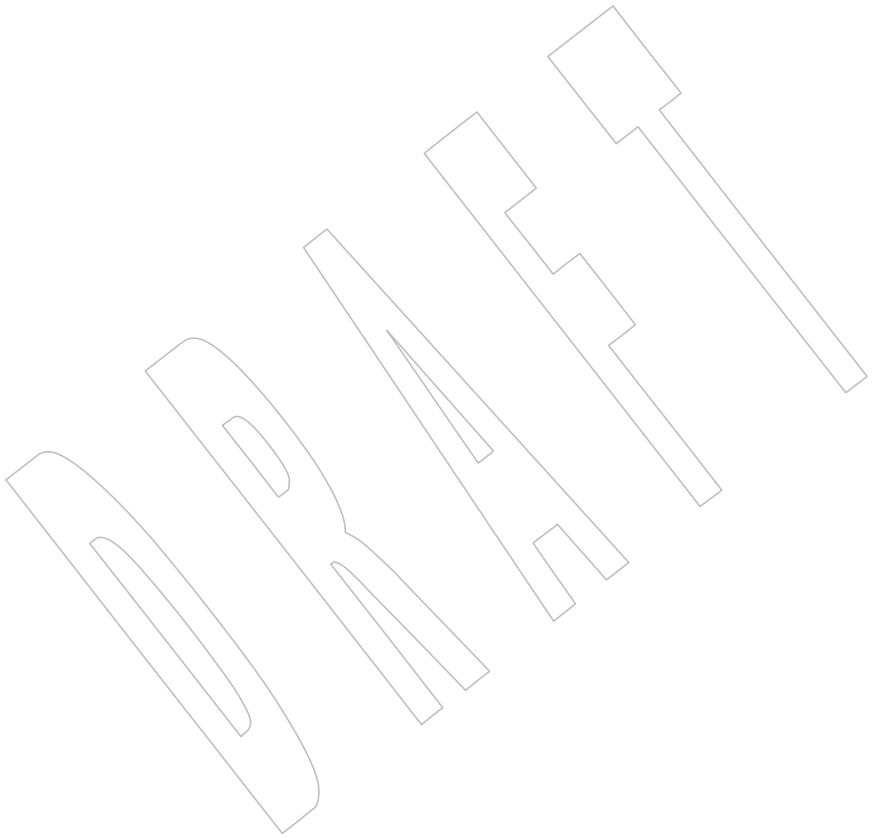
48543 **SYNOPSIS**

48544
```
#include <math.h>
```

48545
```
long double tanl(long double x);
```

48546 **DESCRIPTION**

48547 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 1003.1-200x, 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.
- 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 TCOOFF, 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 1003.1-200x, 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 1003.1-200x, Chapter 11, General Terminal Interface, **<termios.h>**, **<unistd.h>**

CHANGE HISTORY

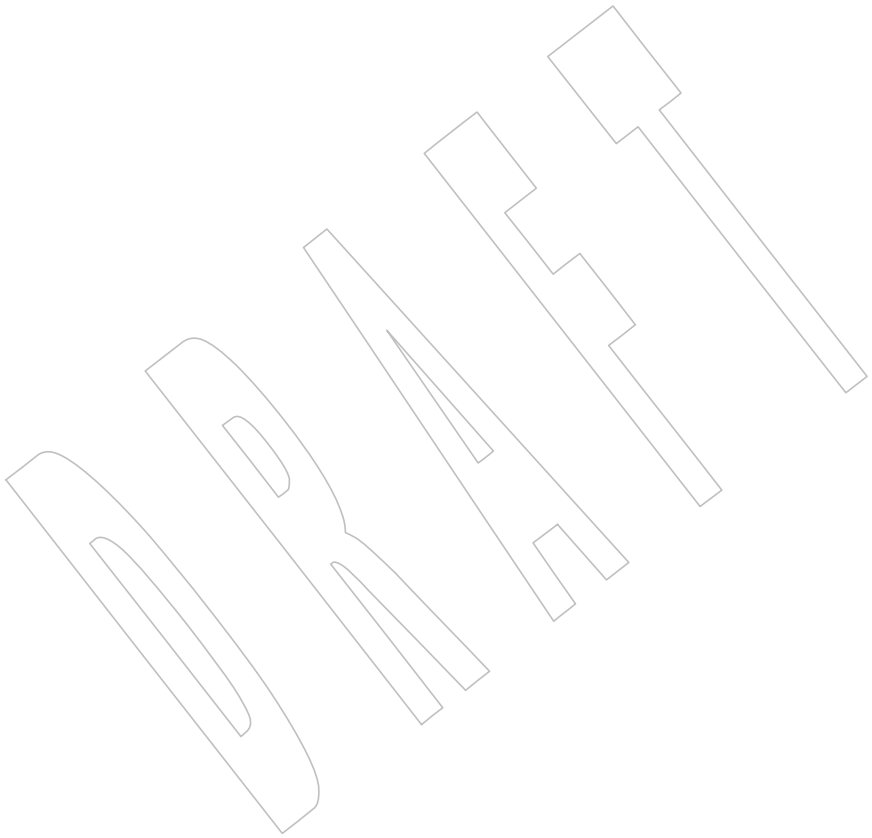
First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

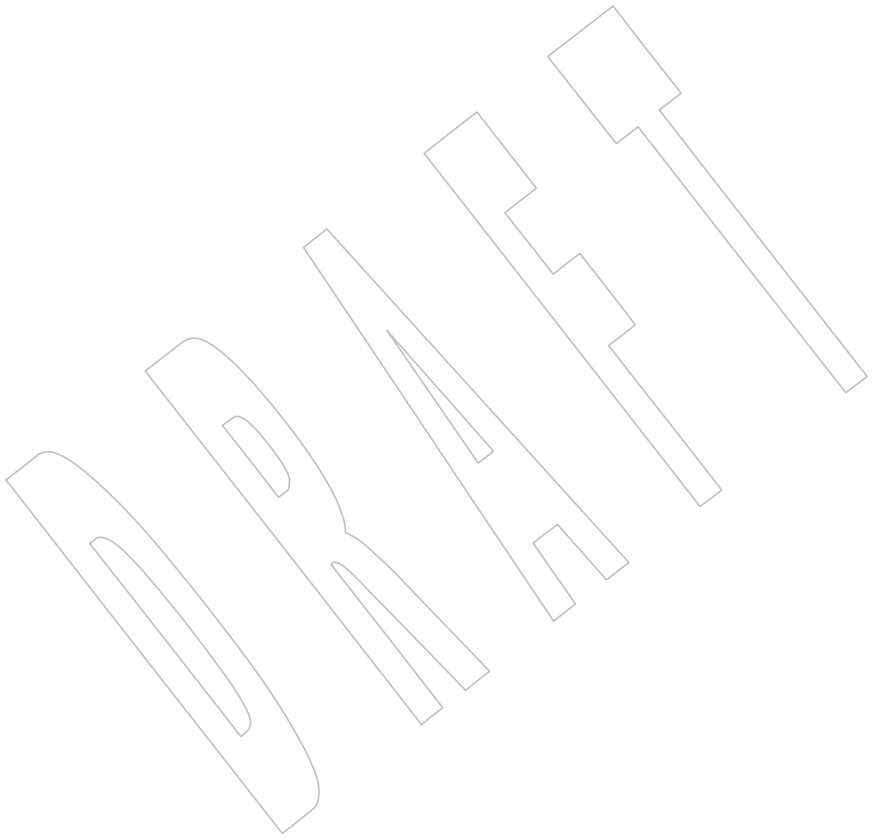
Issue 6

The 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.





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 *tcsetattr*( ) with a call to *tcgetattr*( ) and compare the appropriate field values.

FUTURE DIRECTIONS

None.

SEE ALSO

tcsetattr( ), the Base Definitions volume of IEEE Std 1003.1-200x, 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

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 1003.1-200x, **<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.

48788 • In the DESCRIPTION, text previously conditional on support for _POSIX_JOB_CONTROL
48789 is now mandatory. This is a FIPS requirement.

NAME

tcgetsid — get the process group ID for the session leader for the controlling terminal

SYNOPSIS

```
#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 1003.1-200x, **<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.

Issue 7

The *tcgetsid*( ) function is moved from the XSI option to the Base.

NAME

tcsendbreak — send a break for a specific duration

SYNOPSIS

```
#include <termios.h>

int tcsendbreak(int fildes, int duration);
```

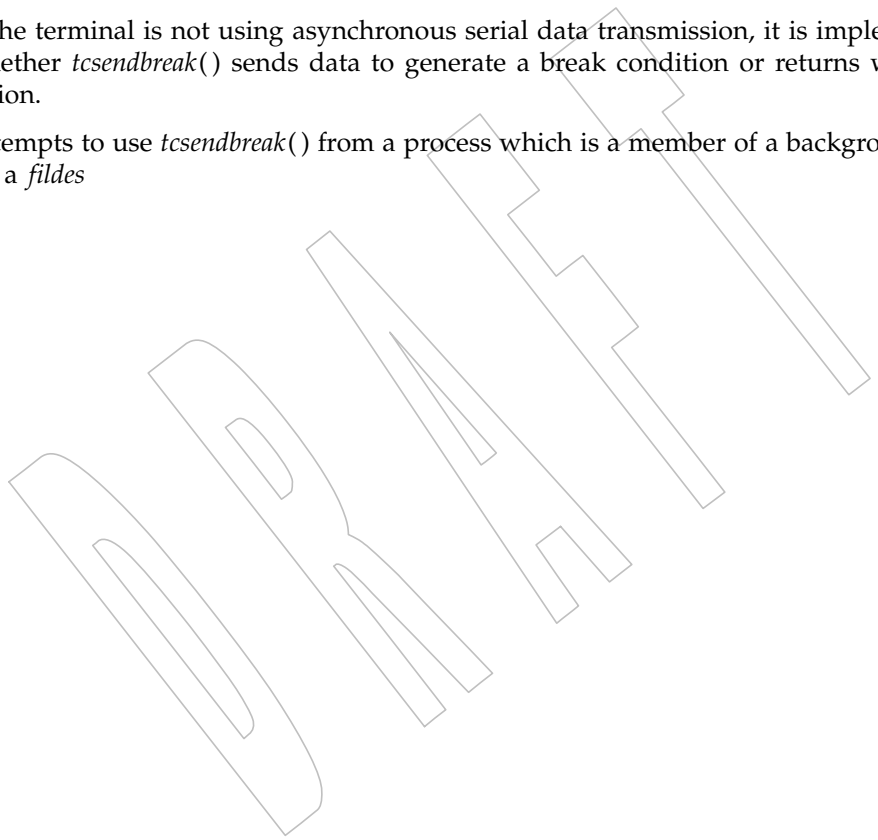
DESCRIPTION

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*



48864 **CHANGE HISTORY**

48865 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

48866 **Issue 6**

48867 The following new requirements on POSIX implementations derive from alignment with the
48868 Single UNIX Specification:

48869 • In the DESCRIPTION, text previously conditional on _POSIX_JOB_CONTROL is now
48870 mandated. This is a FIPS requirement.

48871 • The [EIO] error is added.

NAME

tcsetattr — set the parameters associated with the terminal

SYNOPSIS

```
#include <termios.h>

int tcsetattr(int fildes, int optional_actions,
    const struct termios *termios_p);
```

DESCRIPTION

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 1003.1-200x between the call to *tcgetattr*( ) and *tcsetattr*( ), leaving all other fields and flags unmodified.

No actions defined by this volume of IEEE Std 1003.1-200x, 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 1003.1-200x 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.

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.

[EINTR] A signal interrupted *tcsetattr*( ).

[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.

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 1003.1-200x.

SEE ALSO

cfgetispeed( ), *tcgetattr*( ), the Base Definitions volume of IEEE Std 1003.1-200x, 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 1003.1-200x, **<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 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 normative text 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

XSI

```
#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;
```


(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 *tdelete*( ), *tfind*( ), and *tsearch*( ) if *rootp* is a null pointer on entry.

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 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.

```
#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 *);

    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);
        /* 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",
                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 1003.1-200x, **<search.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.

Issue 6

The normative text 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

fdopendir( ), *readdir*( ), *seekdir*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<dirent.h>**

CHANGE HISTORY

First released in Issue 2.

NAME

tempnam — create a name for a temporary file

SYNOPSIS

OB XSI
```
#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.

Applications should use the *tmpfile*( ), *mkdtemp*( ), or *mkstemp*( ) functions instead of the obsolescent *tempnam*( ) function.

RATIONALE

None.

FUTURE DIRECTIONS

The *tempnam*( ) function may be removed in a future version.

SEE ALSO

fopen( ), *free*( ), *open*( ), *tmpfile*( ), *tmpnam*( ), *unlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

Issue 7

The *tempnam*( ) function is marked obsolescent.

NAME

tfind — search binary search tree

SYNOPSIS

XSI

```
#include <search.h>

void *tfind(const void *key, void *const *rootp,
    int (*compar)(const void *, const void *));
```

DESCRIPTION

Refer to *tdelete( )*.

NAME

tgamma, tgammaf, tgammal — compute gamma( ) function

SYNOPSIS

```
#include <math.h>

double tgamma(double x);
float tgammaf(float x);
long double tgammal(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 1003.1-200x 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 *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 *Gamma*(*x*).

CX If *x* is a negative integer, a domain error may occur and either a NaN (if supported) or an
MX implementation-defined value shall be returned. On systems that support the IEC 60559 Floating-Point option, a domain error shall occur and a NaN shall be returned.

If *x* is ±0, *tgamma*( ), *tgammaf*( ), and *tgammal*( ) shall return ±HUGE_VAL, ±HUGE_VALF, and
MX ±HUGE_VALL, respectively. On systems that support the IEC 60559 Floating-Point option, a pole error shall occur;
CX otherwise, a pole error may occur.

If the correct value would cause overflow, a range error shall occur and *tgamma*( ), *tgammaf*( ), and *tgammal*( ) shall return ±HUGE_VAL, ±HUGE_VALF, or ±HUGE_VALL, respectively, with the same sign as the correct value of the function.

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

If *x* is +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:

MX **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.

MX **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_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:

Domain Error — The value of *x* is a negative 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.

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

For IEEE Std 754-1985 **double**, overflow happens when $0 < x < 1/\text{DBL\_MAX}$, and $171.7 < x$.

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

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 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/65 is applied, correcting the third paragraph in the RETURN VALUE section.

Issue 7

ISO/IEC 9899: 1999 standard, Technical Corrigendum 2 #52 (SD5-XSH-ERN-85) is applied.

NAME

time — get time

SYNOPSIS

```
#include <time.h>

time_t time(time_t *tloc);
```

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 1003.1-200x defers to the ISO C standard.

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:

```
Wed Jun 26 10:32:15 1996
835810335 secs since the Epoch
```

Timing an Event

The following example gets the current time, prints it out in the user's format, and prints the number of minutes to an event being timed.

```
#include <time.h>
#include <stdio.h>
...
time_t now;
int minutes_to_event;
...
time(&now);
minutes_to_event = ...;
printf("The time is ");
puts(asctime(localtime(&now)));
printf("There are %d minutes to the event.\n",
    minutes_to_event);
...
```

APPLICATION USAGE

None.

RATIONALE

The *time*( ) function returns a value in seconds (type **time_t**) while *times*( ) returns a set of values in clock ticks (type **clock_t**). Some historical implementations, such as 4.3 BSD, have mechanisms capable of returning more precise times (see below). A generalized timing scheme to unify these various timing mechanisms has been proposed but not adopted.

Implementations in which **time_t** is a 32-bit signed integer (many historical implementations) fail in the year 2038. IEEE Std 1003.1-200x 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 1003.1-200x 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 1003.1-200x, **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 1003.1-200x 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 1003.1-200x, **<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

SYNOPSIS

CX
```
#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 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
MON 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*.

CPT If _POSIX_CPUTIME is defined, implementations shall support *clock_id* values representing the CPU-time clock of the calling process.

TCT If _POSIX_THREAD_CPUTIME is defined, implementations shall support *clock_id* values representing the CPU-time clock of the calling thread.

CPT | TCT 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.

TSA If *evp–>sigev_sigev_notify* is SIGEV_THREAD and *sev–>sigev_notify_attributes* is not NULL, if the attribute pointed to by *sev–>sigev_notify_attributes* has a thread stack address specified by a call to *pthread_attr_setstack*( ), the results are unspecified if the signal is generated more than once.

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.

CPT | TCT [ENOTSUP] 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

If a timer is created which has *evp−>sigev_sigev_notify* set to SIGEV_THREAD and the attribute pointed to by *evp−>sigev_notify_attributes* has a thread stack address specified by a call to *pthread_attr_setstack*( ), the memory dedicated as a thread stack cannot be recovered. The reason for this is that the threads created in response to a timer expiration are created detached, or in an unspecified way if the thread attribute's *detachstate* is PTHREAD_CREATE_JOINABLE. In neither case is it valid to call *pthread_join*( ), which makes it impossible to determine the lifetime of the created thread which thus means the stack memory cannot be reused.

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_id*s 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 1003.1-200x, **<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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/138 is applied, updating the DESCRIPTION and APPLICATION USAGE sections to describe the case when a timer is created with the notification method set to SIGEV_THREAD.

Issue 7

The *timer_create*( ) function is moved from the Timers option to the Base.

NAME

timer_delete — delete a per-process timer

SYNOPSIS

CX
```
#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 may 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 1003.1-200x, **<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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/139 is applied, updating the ERRORS section so that the [EINVAL] error becomes optional.

Issue 7

The *timer_delete*( ) function is moved from the Timers option to the Base.

NAME

timer_getoverrun, timer_gettime, timer_settime — per-process timers

SYNOPSIS

CX
```
#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, 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, 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_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.

These functions may 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 may fail if:

[EINVAL] The *it_interval* member of *value* is not zero and the timer was created with notification by creation of a new thread (*sigev_sigev_notify* was SIGEV_THREAD) and a fixed stack address has been set in the thread attribute pointed to by *sigev_notify_attributes*.

EXAMPLES

None.

APPLICATION USAGE

Using fixed stack addresses is problematic when timer expiration is signalled by the creation of a new thread. Since it cannot be assumed that the thread created for one expiration is finished before the next expiration of the timer, it could happen that two threads use the same memory as a stack at the same time. This is invalid and produces undefined results.

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 1003.1-200x 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 1003.1-200x, **<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.

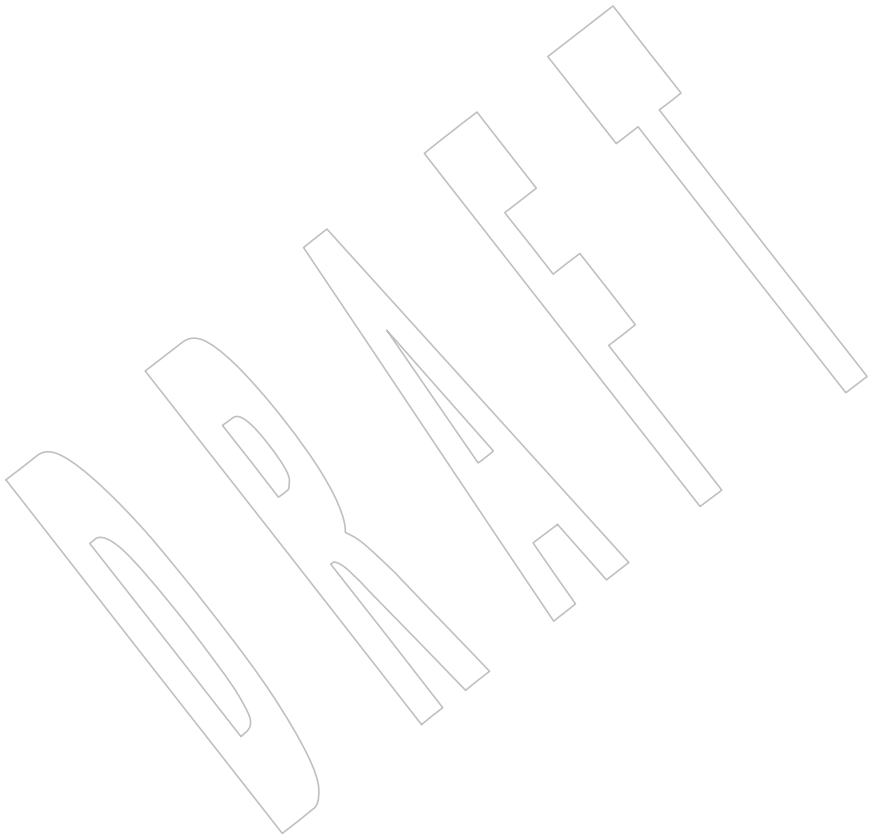
IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/140 is applied, updating the ERRORS section so that the mandatory [EINVAL] error ("The *timerid* argument does not correspond to an ID returned by *timer_create*( ) but not yet deleted by *timer_delete*( )") becomes optional.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/141 is applied, updating the ERRORS section to include an optional [EINVAL] error for the case when a timer is created with the notification method set to SIGEV_THREAD. APPLICATION USAGE text is also added.

Issue 7

The *timer_getoverrun*( ), *timer_gettime*( ), and *timer_settime*( ) functions are moved from the Timers option to the Base.

Functionality relating to the Realtime Signals Extension option is moved to the Base.



```
...
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);

    fputs(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 *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 "realtime" in *alarm( )*.

If the type **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 1003.1-200x 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 "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

alarm( ), *exec*, *fork( )*, *sysconf( )*, *time( )*, *wait( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/times.h>**

CHANGE HISTORY

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

NAME

timezone — difference from UTC and local standard time

SYNOPSIS

XSI

```
#include <time.h>

extern long timezone;
```

DESCRIPTION

Refer to *tzset( )*.

NAME

tmpfile — create a temporary file

SYNOPSIS

```
#include <stdio.h>

FILE *tmpfile(void);
```

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 1003.1-200x 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+*), except that implementations may restrict the permissions, either by clearing the file mode bits or setting them to the value S_IRUSR | S_IWUSR.

CX In some implementations, a permanent file may be left behind if the process calling *tmpfile*( ) is killed while it is processing a call to *tmpfile*( ).

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 CX created. Otherwise, it shall return a null pointer and set *errno* to indicate the error.

ERRORS

The *tmpfile*( ) function shall fail if:

CX [EINTR] A signal was caught during *tmpfile*( ).

CX [EMFILE] All file descriptors available to the process are currently open.

CX [ENFILE] The maximum allowable number of files is currently open in the system.

CX [ENOSPC] The directory or file system which would contain the new file cannot be expanded.

CX [EOVERFLOW] The file is a regular file and the size of the file cannot be represented correctly in an object of type **off_t**.

The *tmpfile*( ) function may fail if:

CX [EMFILE] {FOPEN_MAX} streams are currently open in the calling process.

CX [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.

```
#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( ), *mkdtemp*( ), *tmpnam*( ), *unlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

Issue 7

SD5-XBD-ERN-4 is applied, changing the definition of the [EMFILE] error.

Austin Group Interpretation 1003.1-2001 #025 is applied, clarifying that implementations may restrict the permissions of the file created.

NAME

tmpnam — create a name for a temporary file

SYNOPSIS

OB
```
#include <stdio.h>

char *tmpnam(char *s);
```

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 1003.1-200x 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 1003.1-200x, except *tempnam*( ), calls *tmpnam*( ).

CX If the application uses any of the POSIX threads functions, 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 **char**s; *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.

Applications should use the *tmpfile*( ), *mkstemp*( ), or *mkdtemp*( ) functions instead of the obsolescent *tmpnam*( ) function.

RATIONALE

None.

FUTURE DIRECTIONS

The *tmpnam*( ) function may be removed in a future version.

SEE ALSO

fopen( ), *open*( ), *mkdtemp*( ), *tempnam*( ), *tmpfile*( ), *unlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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 normative text 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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/142 is applied, updating the DESCRIPTION to allow implementations of the *tempnam*( ) function to call *tmpnam*( ).

Issue 7

The *tmpnam*( ) function is marked obsolescent.

NAME

toascii — translate an integer to a 7-bit ASCII character

SYNOPSIS

OB 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

The *toascii*( ) function cannot be used portably in a localized application.

RATIONALE

None.

FUTURE DIRECTIONS

The *toascii*( ) function may be removed in a future version.

SEE ALSO

isascii( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**

CHANGE HISTORY

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

Issue 7

The *toascii*( ) function is marked obsolescent.

NAME

tolower, tolower_l — transliterate uppercase characters to lowercase

SYNOPSIS

```
#include <ctype.h>

int tolower(int c);
CX int tolower_l(int c, locale_t locale);
```

DESCRIPTION

CX For *tolower*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

CX The *tolower*( ) and *tolower_l*( ) functions have 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 CX behavior is undefined. If the argument of *tolower*( ) or *tolower_l*( ) represents an uppercase letter, and there exists a corresponding lowercase letter as defined by character type information in the CX program locale or in the locale represented by *locale*, respectively (category *LC_CTYPE*), the result shall be the corresponding lowercase letter. All other arguments in the domain are returned unchanged.

RETURN VALUE

CX Upon successful completion, the *tolower*( ) and *tolower_l*( ) functions shall return the lowercase letter corresponding to the argument passed; otherwise, they shall return the argument unchanged.

ERRORS

The *tolower_l*( ) function may fail if:

CX [EINVAL] *locale* is not a valid locale object handle.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

setlocale( ), *uselocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale, **<ctype.h>**, **<locale.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.

Issue 7

The *tolower_l*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

NAME

toupper, toupper_l — transliterate lowercase characters to uppercase

SYNOPSIS

```
#include <ctype.h>

int toupper(int c);
CX int toupper_l(int c, locale_t locale);
```

DESCRIPTION

CX For *toupper*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

CX The *toupper*( ) and *toupper_l*( ) functions have 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 CX behavior is undefined. If the argument of *toupper*( ) or *toupper_l*( ) represents a lowercase letter, and there exists a corresponding uppercase letter as defined by character type information in the CX program locale or in the locale represented by *locale*, respectively (category *LC_CTYPE*), the result shall be the corresponding uppercase letter. All other arguments in the domain are returned unchanged.

RETURN VALUE

CX Upon successful completion, *toupper*( ) and *toupper_l*( ) shall return the uppercase letter corresponding to the argument passed.

ERRORS

The *toupper_l*( ) function may fail if:

CX [EINVAL] *locale* is not a valid locale object handle.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

setlocale( ), *uselocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale, **<ctype.h>**, **<locale.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.

Issue 7

The *toupper_l*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

NAME

towctrans, towctrans_l — wide-character transliteration

SYNOPSIS

```
#include <wctype.h>

wint_t towctrans(wint_t wc, wctrans_t desc);
CX wint_t towctrans_l(wint_t wc, wctrans_t desc,
       locale_t locale);
```

DESCRIPTION

CX For *towctrans*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

CX The *towctrans*( ) and *towctrans_l*( ) functions shall transliterate the wide-character code *wc* using the mapping described by *desc*.

CX The current setting of the *LC_CTYPE* category in the current locale of the process or in the locale CX represented by *locale*, respectively, should be the same as during the call to *wctrans*( ) or *wctrans_l*( ) 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 If the value of *desc* is invalid (that is, not obtained by a call to *wctrans_l*( ) with the same locale object *locale*) the result is unspecified.

CX An application wishing to check for error situations should set *errno* to 0 before calling *towctrans*( ) or *towctrans_l*( ).

If *errno* is non-zero on return, an error has occurred.

RETURN VALUE

CX If successful, the *towctrans*( ) and *towctrans_l*( ) functions shall return the mapped value of *wc* using the mapping described by *desc*. Otherwise, they shall return *wc* unchanged.

ERRORS

These functions may fail if:

CX [EINVAL] *desc* contains an invalid transliteration descriptor.

The *towctrans_l*( ) function may fail if:

CX [EINVAL] *locale* is not a valid locale object handle.

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"))
towlower_l(wc, locale)    towctrans_l(wc, wctrans("tolower"), locale)
towupper(wc)              towctrans(wc, wctrans("toupper"))
towupper_l(wc, locale)    towctrans_l(wc, wctrans("toupper"), locale)
```

50068 **RATIONALE**
50069 None.

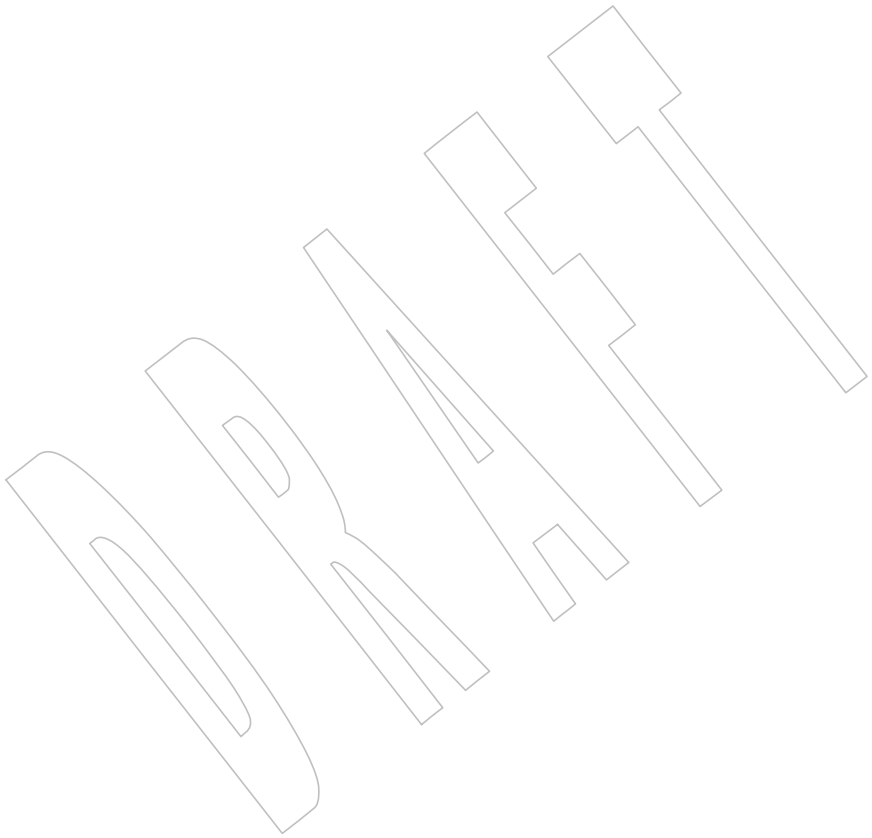
50070 **FUTURE DIRECTIONS**
50071 None.

50072 **SEE ALSO**
50073 *towlower( )*, *towupper( )*, *wctrans( )*, the Base Definitions volume of IEEE Std 1003.1-200x,
50074 **<wctype.h>**

50075 **CHANGE HISTORY**
50076 First released in Issue 5. Derived from ISO/IEC 9899: 1990/Amendment 1: 1995 (E).

50077 **Issue 6**
50078 Extensions beyond the ISO C standard are marked.

50079 **Issue 7**
50080 The *towctrans_l( )* function is added from The Open Group Technical Standard, 2006, Extended
50081 API Set Part 4.



50082 **NAME**

50083 towlower, towlower_l — transliterate uppercase wide-character code to lowercase

50084 **SYNOPSIS**

50085
```
#include <wctype.h>
```

50086
```
wint_t towlower(wint_t wc);
```

50087 CX
```
wint_t towlower_l(wint_t wc, locale_t locale);
```

50088 **DESCRIPTION**

50089 CX For *towlower*( ): The functionality described on this reference page is aligned with the ISO C
50090 standard. Any conflict between the requirements described here and the ISO C standard is
50091 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

50092 CX The *towlower*( ) and *towlower_l*( ) functions have as a domain a type **wint_t**, the value of which
50093 the application shall ensure is a character representable as a **wchar_t**, and a wide-character code
50094 corresponding to a valid character in the current locale or the value of WEOF. If the argument
50095 CX has any other value, the behavior is undefined. If the argument of *towlower*( ) or *towlower_l*( )
50096 represents an uppercase wide-character code, and there exists a corresponding lowercase wide-
50097 CX character code as defined by character type information in the locale of the process or in the
50098 locale represented by *locale*, respectively (category *LC_CTYPE*), the result shall be the
50099 corresponding lowercase wide-character code. All other arguments in the domain are returned
50100 unchanged.

50101 **RETURN VALUE**

50102 CX Upon successful completion, the *towlower*( ) and *towlower_l*( ) functions shall return the
50103 lowercase letter corresponding to the argument passed; otherwise, they shall return the
50104 argument unchanged.

50105 **ERRORS**

50106 The *towlower_l*( ) function may fail if:

50107 CX [EINVAL] *locale* is not a valid locale object handle.

50108 **EXAMPLES**

50109 None.

50110 **APPLICATION USAGE**

50111 None.

50112 **RATIONALE**

50113 None.

50114 **FUTURE DIRECTIONS**

50115 None.

50116 **SEE ALSO**

50117 *setlocale*( ), *uselocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale,
50118 **<locale.h>**, **<wctype.h>**

50119 **CHANGE HISTORY**

50120 First released in Issue 4.

50121 **Issue 5**

50122 The following change has been made in this issue for alignment with
50123 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 normative text is updated to avoid use of the term "must" for application requirements.

Issue 7

The *towlower_l*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

NAME

towupper, towupper_l — transliterate lowercase wide-character code to uppercase

SYNOPSIS

```
#include <wctype.h>

wint_t towupper(wint_t wc);
CX wint_t towupper_l(wint_t wc, locale_t locale);
```

DESCRIPTION

CX For *towupper*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

CX The *towupper*( ) and *towupper_l*( ) functions have 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 CX has any other value, the behavior is undefined. If the argument of *towupper*( ) or *towupper_l*( ) represents a lowercase wide-character code, and there exists a corresponding uppercase wide- CX character code as defined by character type information in the locale of the process or in the locale represented by *locale*, respectively (category *LC_CTYPE*), the result shall be the corresponding uppercase wide-character code. All other arguments in the domain are returned unchanged.

RETURN VALUE

CX Upon successful completion, the *towupper*( ) and *towupper_l*( ) functions shall return the uppercase letter corresponding to the argument passed. Otherwise, they shall return the argument unchanged.

ERRORS

The *towupper_l*( ) function may fail if:

CX [EINVAL] *locale* is not a valid locale object handle.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

setlocale( ), *uselocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale, **<locale.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 normative text is updated to avoid use of the term "must" for application requirements.

Issue 7

The *towupper_l*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

50180 **NAME**

50181 trunc, truncf, truncl — round to truncated integer value

50182 **SYNOPSIS**

50183
```
#include <math.h>
```

50184
```
double trunc(double x);
```
50185
```
float truncf(float x);
```
50186
```
long double truncl(long double x);
```

50187 **DESCRIPTION**

50188 CX The functionality described on this reference page is aligned with the ISO C standard. Any
50189 conflict between the requirements described here and the ISO C standard is unintentional. This
50190 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

50191 These functions shall round their argument to the integer value, in floating format, nearest to but
50192 no larger in magnitude than the argument.

50193 **RETURN VALUE**

50194 Upon successful completion, these functions shall return the truncated integer value.

50195 MX If *x* is NaN, a NaN shall be returned.

50196 If *x* is ±0 or ±Inf, *x* shall be returned.

50197 **ERRORS**

50198 No errors are defined.

50199 **EXAMPLES**

50200 None.

50201 **APPLICATION USAGE**

50202 None.

50203 **RATIONALE**

50204 None.

50205 **FUTURE DIRECTIONS**

50206 None.

50207 **SEE ALSO**

50208 The Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

50209 **CHANGE HISTORY**

50210 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

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.

[EFBIG] 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:

[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

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

open( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

Issue 6

This reference page is split out from the *ftruncate*( ) reference page.

The normative text 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.

Issue 7

The *truncate*( ) function is moved from the XSI option to the Base.

50277 **NAME**

50278 truncf, truncl — round to truncated integer value

50279 **SYNOPSIS**

50280
```
#include <math.h>

float truncf(float x);
```
50281
```
long double truncl(long double x);
```
50282

50283 **DESCRIPTION**

50284 Refer to *trunc( )*.

NAME

tsearch — search a binary search tree

SYNOPSIS

XSI

```
#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 thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

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 file associated with the *fildes* argument is not a terminal.

The *ttyname_r*( ) function may fail if:

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

[ENOTTY] The file associated with the *fildes* argument is not 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.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

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

Issue 5

The *ttyname_r*( ) function is included for alignment with the POSIX Threads Extension.

A note indicating that the *ttyname*( ) function need not be reentrant is added to the DESCRIPTION.

Issue 6

The *ttyname_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.

Issue 7

SD5-XSH-ERN-100 is applied, correcting the definition of the [ENOTTY] error condition.

The *ttyname_r*( ) function is moved from the Thread-Safe Functions option to the Base.

50352 **NAME**

50353 twalk — traverse a binary search tree

50354 **SYNOPSIS**

50355 XSI
```
#include <search.h>

void twalk(const void *root,
    void (*action)(const void *, VISIT, int ));
```

50358 **DESCRIPTION**

50359 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 1003.1-200x, 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:

TZ	*timezone*
EST5EDT	5*60*60
GMT0	0*60*60
JST-9	–9*60*60
MET-1MEST	–1*60*60
MST7MDT	7*60*60
PST8PDT	8*60*60

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

50400 **SEE ALSO**

50401 *ctime( )*, *localtime( )*, *mktime( )*, *strftime( )*, the Base Definitions volume of IEEE Std 1003.1-200x,
50402 Chapter 8, Environment Variables, **<time.h>**

50403 **CHANGE HISTORY**

50404 First released in Issue 1. Derived from Issue 1 of the SVID.

50405 **Issue 6**

50406 The example is corrected.

NAME

ulimit — get and set process limits

SYNOPSIS

OB XSI
```
#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

Since the *ulimit*( ) function uses type **long** rather than **rlim_t**, this function is not sufficient for file sizes on many current systems. Applications should use the *getrlimit*( ) or *setrlimit*( ) functions instead of the obsolescent *ulimit*( ) function.

RATIONALE

None.

FUTURE DIRECTIONS

The *ulimit*( ) function may be removed in a future version.

SEE ALSO

exec, *getrlimit*( ), *setrlimit*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ulimit.h>**

CHANGE HISTORY

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

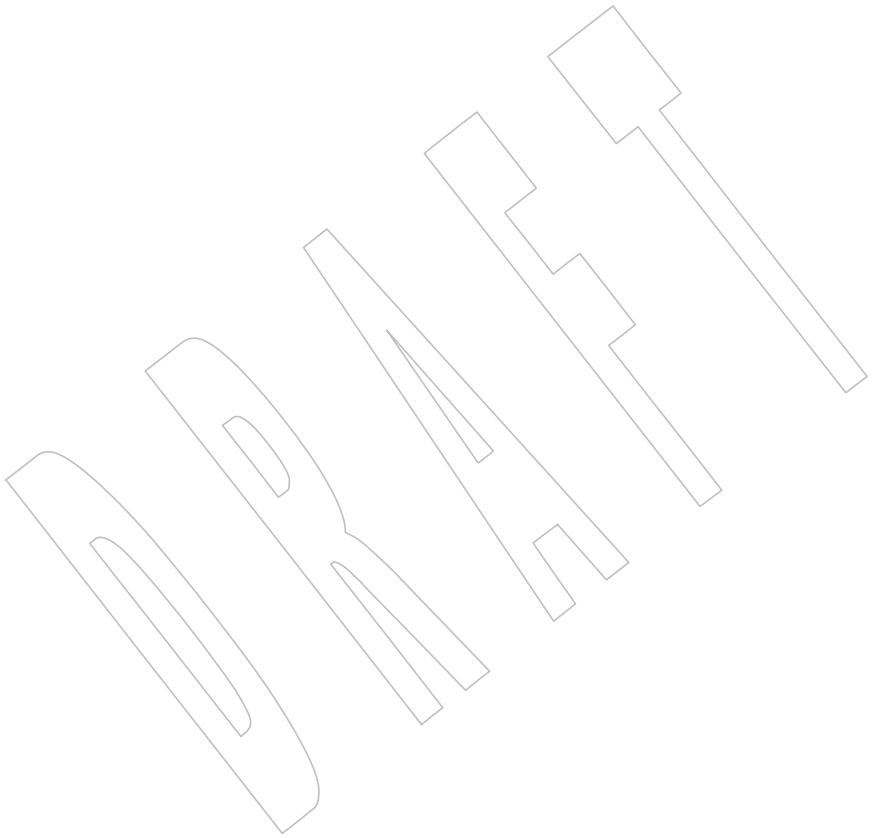
Issue 5

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.

Issue 7

The *ulimit*( ) function is marked obsolescent.



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 file mode creation mask of the process 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 file mode creation mask of the process is used to turn off permission bits in the *mode* argument supplied during calls to the following functions:

- *open*( ), *openat*( ), *creat*( ), *mkdir*( ), *mkdirat*( ), *mkfifo*( ), and *mkfifoat*( )
- XSI *mknod*( ), *mknodat*( )
- MSG *mq_open*( )
- *sem_open*( )

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 IEEE Std 1003.1-200x.

FUTURE DIRECTIONS

None.

SEE ALSO

creat( ), *exec*, *mkdir*( ), *mkfifo*( ), *mknod*( ), *mq_open*( ), *open*( ), *sem_open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/143 is applied, adding the *mknod*( ), *mq_open*( ), and *sem_open*( ) functions to the DESCRIPTION and SEE ALSO sections.

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 1003.1-200x 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 1003.1-200x 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 1003.1-200x, **<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

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 1003.1-200x 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 1003.1-200x, **<stdio.h>**

50596 **CHANGE HISTORY**
50597 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

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 1003.1-200x 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:

CX [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.

50639 **SEE ALSO**

50640 *fseek( )*, *fsetpos( )*, *read( )*, *rewind( )*, *setbuf( )*, the Base Definitions volume of IEEE Std 1003.1-200x,
50641 **<stdio.h>**, **<wchar.h>**

50642 **CHANGE HISTORY**

50643 First released in Issue 4. Derived from the MSE working draft.

50644 **Issue 5**

50645 The Optional Header (OH) marking is removed from **<stdio.h>**.

50646 **Issue 6**

50647 The [EILSEQ] optional error condition is marked CX.

NAME

unlink, unlinkat — remove a directory entry relative to directory file descriptor

SYNOPSIS

```
#include <unistd.h>

int unlink(const char *path);
int unlinkat(int fd, const char *path, int flag);
```

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.

The *unlinkat*( ) function shall be equivalent to the *unlink*( ) or *rmdir*( ) function except in the case where *path* specifies a relative path. In this case the directory entry to be removed is determined relative to the directory associated with the file descriptor *fd* instead of the current working directory. It is unspecified whether directory searches are permitted based on whether the file was opened with search permission or on the current permissions of the directory underlying the file descriptor.

Values for *flag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in **<fcntl.h>**:

AT_REMOVEDIR Remove the directory entry specified by *fd* and *path* as a directory, not a normal file.

If *unlinkat*( ) is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior shall be identical to a call to *unlink*( ) or *rmdir*( ) respectively, depending on whether or not the AT_REMOVEDIR bit is set in *flag*.

RETURN VALUE

Upon successful completion, these functions shall return 0. Otherwise, these functions shall return –1 and set *errno* to indicate the error. If –1 is returned, the named file shall not be changed.

ERRORS

These functions 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.

XSI [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.

[EROFS] The directory entry to be unlinked is part of a read-only file system.

The *unlinkat*( ) function shall fail if:

[EBADF] The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

[EEXIST] or [ENOTEMPTY]
The *flag* parameter has the AT_REMOVEDIR bit set and 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.

[ENOTDIR] The *flag* parameter has the AT_REMOVEDIR bit set and *path* does not name a directory.

These functions may fail and not unlink the file if:

XSI [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.

The *unlinkat*( ) function may fail if:

[EINVAL] The value of the *flag* argument is not valid.

[ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

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**.

```
#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**.

```
#include <sys/types.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <sys/stat.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) {
    close(pfd);
    unlink(LOCKFILE);
    exit(1);
}
```

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.

```
#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\n", 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 1003.1-200x 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.

The standard developers reviewed TR 24715-2006 and noted that LSB-conforming implementations may return [EISDIR] instead of [EPERM] when unlinking a directory. A change to permit this behavior by changing the requirement for [EPERM] to [EPERM] or [EISDIR] was considered, but decided against since it would break existing strictly conforming and conforming applications. Applications written for portability to both this standard and the LSB should be prepared to handle either error code.

The purpose of the *unlinkat*( ) function is to remove directory entries in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to *unlink*( ), resulting in unspecified behavior. By opening a file descriptor for the target directory and using the *unlinkat*( ) function it can be guaranteed that the removed directory entry is located relative to the desired directory.

50822 **FUTURE DIRECTIONS**

50823 None.

50824 **SEE ALSO**

50825 *close( )*, *link( )*, *remove( )*, *rename( )*, *rmdir( )*, *symlink( )*, the Base Definitions volume of
50826 IEEE Std 1003.1-200x, **<fcntl.h>**, **<unistd.h>**

50827 **CHANGE HISTORY**

50828 First released in Issue 1. Derived from Issue 1 of the SVID.

50829 **Issue 5**

50830 The [EBUSY] error is added to the optional part of the ERRORS section.

50831 **Issue 6**

50832 The following new requirements on POSIX implementations derive from alignment with the
50833 Single UNIX Specification:

50834 • In the DESCRIPTION, the effect is specified if *path* specifies a symbolic link.

50835 • The [ELOOP] mandatory error condition is added.

50836 • A second [ENAMETOOLONG] is added as an optional error condition.

50837 • The [ETXTBSY] optional error condition is added.

50838 The following changes were made to align with the IEEE P1003.1a draft standard:

50839 • The [ELOOP] optional error condition is added.

50840 The normative text is updated to avoid use of the term "must" for application requirements.

50841 **Issue 7**

50842 Text arising from the LSB Conflicts TR is added to the RATIONALE about the use of [EPERM]
50843 and [EISDIR].

50844 The *unlinkat( )* function is added from The Open Group Technical Standard, 2006, Extended API
50845 Set Part 2.

50846 **NAME**

50847 unlinkat — remove a directory entry relative to directory file descriptor

50848 **SYNOPSIS**

50849
```
#include <unistd.h>
```

50850
```
int unlinkat(int fd, const char *path, int flag);
```

50851 **DESCRIPTION**

50852 Refer to *unlink( )*.

50853 **NAME**

50854 unlockpt — unlock a pseudo-terminal master/slave pair

50855 **SYNOPSIS**

50856 XSI
```
#include <stdlib.h>

int unlockpt(int fildes);
```

50858 **DESCRIPTION**

50859 The *unlockpt*( ) function shall unlock the slave pseudo-terminal device associated with the master
50860 to which *fildes* refers.

50861 Conforming applications shall ensure that they call *unlockpt*( ) before opening the slave side of a
50862 pseudo-terminal device.

50863 **RETURN VALUE**

50864 Upon successful completion, *unlockpt*( ) shall return 0. Otherwise, it shall return –1 and set *errno*
50865 to indicate the error.

50866 **ERRORS**

50867 The *unlockpt*( ) function may fail if:

50868 [EBADF] The *fildes* argument is not a file descriptor open for writing.

50869 [EINVAL] The *fildes* argument is not associated with a master pseudo-terminal device.

50870 **EXAMPLES**

50871 None.

50872 **APPLICATION USAGE**

50873 None.

50874 **RATIONALE**

50875 None.

50876 **FUTURE DIRECTIONS**

50877 None.

50878 **SEE ALSO**

50879 *grantpt*( ), *open*( ), *ptsname*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

50880 **CHANGE HISTORY**

50881 First released in Issue 4, Version 2.

50882 **Issue 5**

50883 Moved from X/OPEN UNIX extension to BASE.

50884 **Issue 6**

50885 The normative text is updated to avoid use of the term “must” for application requirements.

NAME

unsetenv — remove an environment variable

SYNOPSIS

CX
```
#include <stdlib.h>

int unsetenv(const char *name);
```

DESCRIPTION

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 thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

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 *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.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Refer to the RATIONALE section in *setenv*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

getenv( ), *setenv*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the IEEE P1003.1a draft standard.

NAME

uselocale — use locale in current thread

SYNOPSIS

CX
```
#include <locale.h>

locale_t uselocale(locale_t newloc);
```

DESCRIPTION

The *uselocale*( ) function shall set the current locale for the current thread to the locale represented by *newloc*.

The value for the *newloc* argument shall be one of the following:

1. A value returned by the *newlocale*( ) or *duplocale*( ) functions
2. The special locale object descriptor *LC_GLOBAL_LOCALE*
3. (**locale_t**)0

Once the *uselocale*( ) function has been called to install a thread-local locale, the behavior of every interface using data from the current locale shall be affected for the calling thread. The current locale for other threads shall remain unchanged.

If the *newloc* argument is a null pointer, the object returned is the current locale or *LC_GLOBAL_LOCALE* if there has been no previous call to *uselocale*( ) for the current thread.

If the *newloc* argument is *LC_GLOBAL_LOCALE*, the thread shall use the global locale determined by the *setlocale*( ) function.

RETURN VALUE

The *uselocale*( ) function returns the locale handle from the previous call for the current thread. If there was no such previous call, the function shall return the value *LC_GLOBAL_LOCALE*.

ERRORS

The *uselocale*( ) function may fail if:

[EINVAL] *locale* is not a valid locale object.

EXAMPLES

None.

APPLICATION USAGE

Unlike the *setlocale*( ) function, the *uselocale*( ) function does not allow replacing some locale categories only. Applications that need to install a locale which differs only in a few categories must use *newlocale*( ) to change a locale object equivalent to the currently used locale and install it.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

duplocale( ), *freelocale*( ), *newlocale*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<locale.h>**

50960 **CHANGE HISTORY**
50961 First released in Issue 7.

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 effective user ID of the calling process 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.

[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

utimes( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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 normative text is updated to avoid use of the term "must" for application requirements.

NAME

futimesat, utimes — set file access and modification times relative to directory file descriptor

SYNOPSIS

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

int futimesat(int fd, const char *path, const struct timeval times[2]);
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.

The *futimesat*( ) function shall be equivalent to the *utimes*( ) function except in the case where *path* specifies a relative path. In this case the access and modification time is set to that of a file relative to the directory associated with the file descriptor *fd* instead of the current working directory. It is unspecified whether directory searches are permitted based on whether the file was opened with search permission or on the current permissions of the directory underlying the file descriptor.

If *futimesat*( ) is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior shall be identical to a call to *utimes*( ).

RETURN VALUE

Upon successful completion, these functions shall return 0. Otherwise, these functions shall return –1 and set *errno* to indicate the error. If –1 is returned, the file times shall not be affected.

ERRORS

These functions 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 *futimesat*( ) function shall fail if:

[EBADF] The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

These functions 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}.

The *futimesat*( ) function may fail if:

[ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

EXAMPLES

None.

APPLICATION USAGE

For applications portability, the *utime*( ) function should be used to set file access and modification times instead of *utimes*( ).

RATIONALE

The purpose of the *futimesat*( ) function is to set the access and modification time of files in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to *utimes*( ), resulting in unspecified behavior. By opening a file descriptor for the target directory and using the *futimesat*( ) function it can be guaranteed that the changed file is located relative to the desired directory.

FUTURE DIRECTIONS

None.

SEE ALSO

utime( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

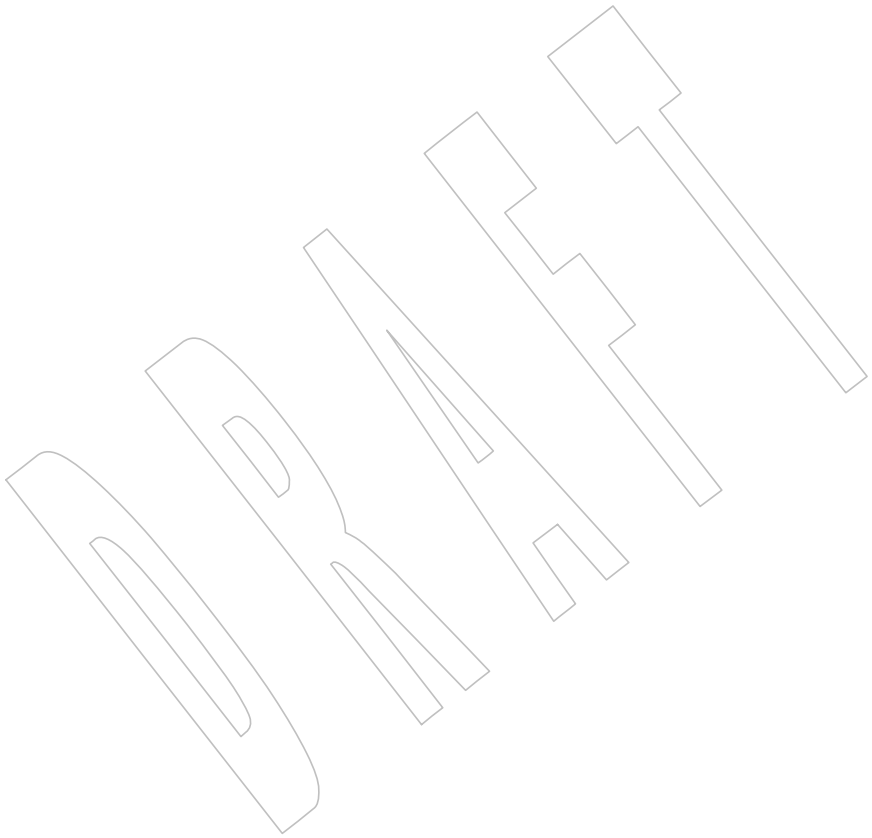
The normative text is updated to avoid use of the term "must" for application requirements.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

Issue 7

The LEGACY marking is removed.

The *futimesat*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 2.



51119 **NAME**

51120 va_arg, va_copy, va_end, va_start — handle variable argument list

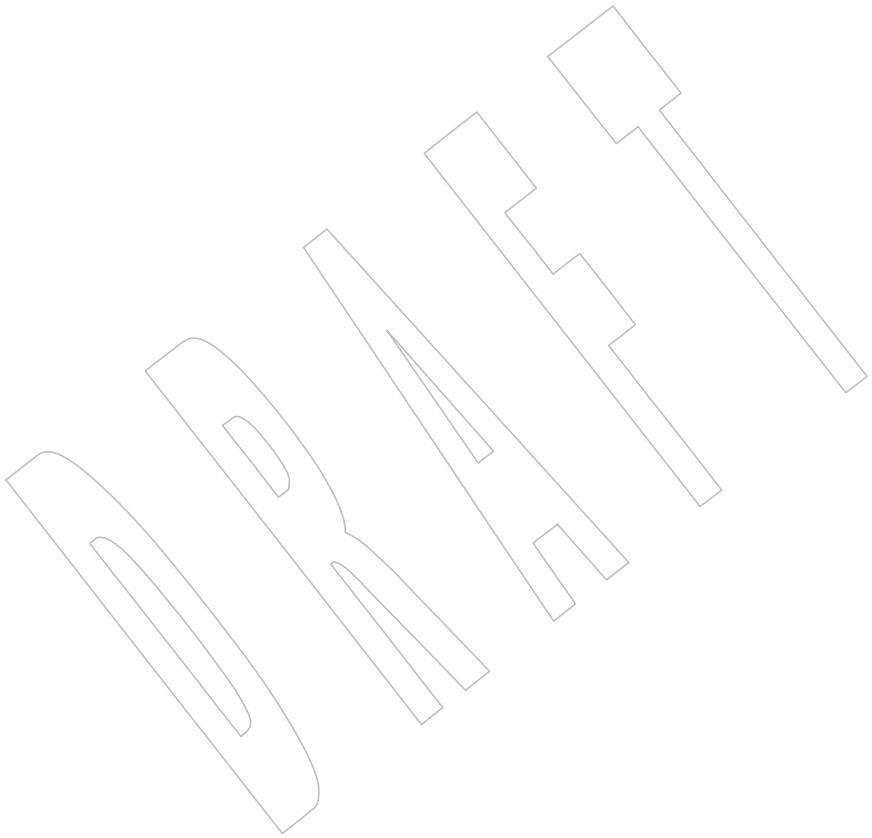
51121 **SYNOPSIS**

51122
```
#include <stdarg.h>
```

51123
```
type va_arg(va_list ap, type);
```
51124
```
void va_copy(va_list dest, va_list src);
```
51125
```
void va_end(va_list ap);
```
51126
```
void va_start(va_list ap, argN);
```

51127 **DESCRIPTION**

51128 Refer to the Base Definitions volume of IEEE Std 1003.1-200x, **<stdarg.h>**.



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

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 1003.1-200x defers to the ISO C standard.

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 1003.1-200x, **<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 *vfprintf*( ), *vprintf*( ), *vsnprintf*( ), and *vsprintf*( ) functions are updated for alignment with the ISO/IEC 9899: 1999 standard.

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

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 1003.1-200x defers to the ISO C standard.

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 1003.1-200x, **<stdarg.h>**, **<stdio.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

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 1003.1-200x 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 1003.1-200x, **<stdarg.h>**, **<stdio.h>**, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

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 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

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 1003.1-200x, **<stdarg.h>**, **<stdio.h>**, **<wchar.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

51283 **NAME**

51284 vprintf — format the output of a stdarg argument list

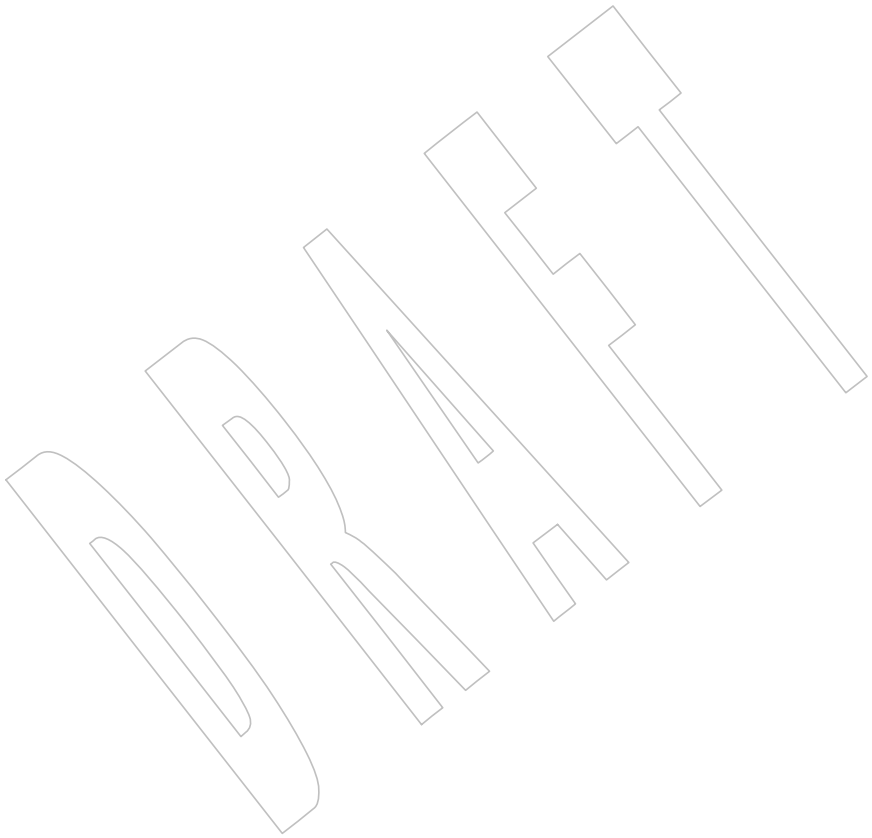
51285 **SYNOPSIS**

51286
```
#include <stdarg.h>
```
51287
```
#include <stdio.h>
```

51288
```
int vprintf(const char *restrict format, va_list ap);
```

51289 **DESCRIPTION**

51290 Refer to *vfprintf( )*.



51291 **NAME**

51292 vscanf — format input of a stdarg argument list

51293 **SYNOPSIS**

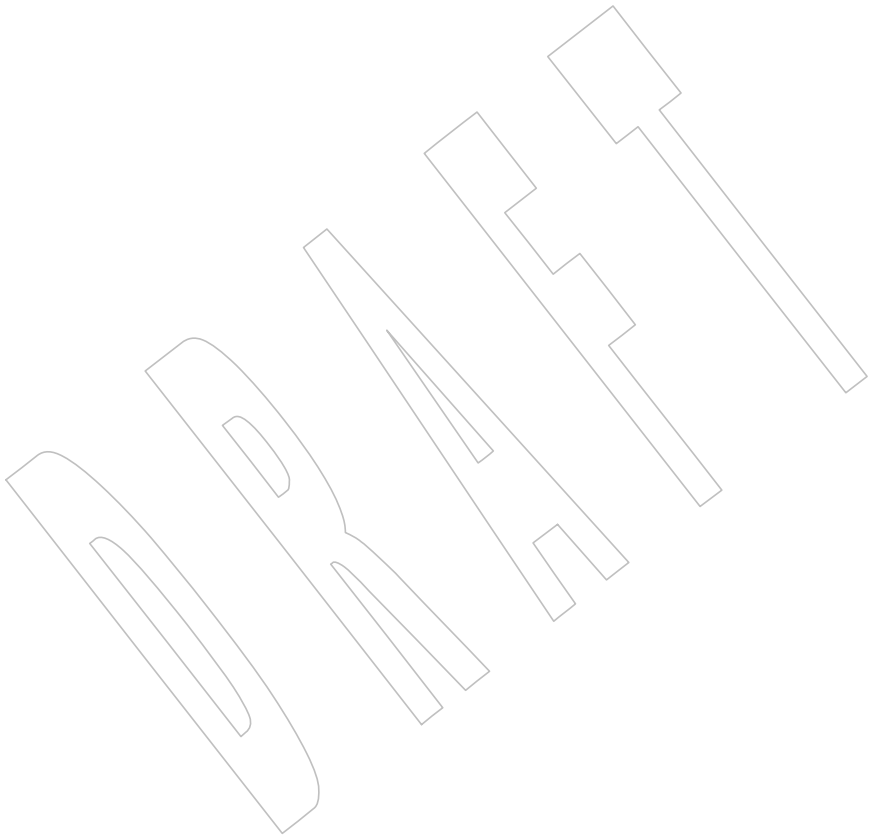
51294 `#include <stdarg.h>`

51295 `#include <stdio.h>`

51296 `int vscanf(const char *restrict format, va_list arg);`

51297 **DESCRIPTION**

51298 Refer to *vfscanf( )*.



NAME

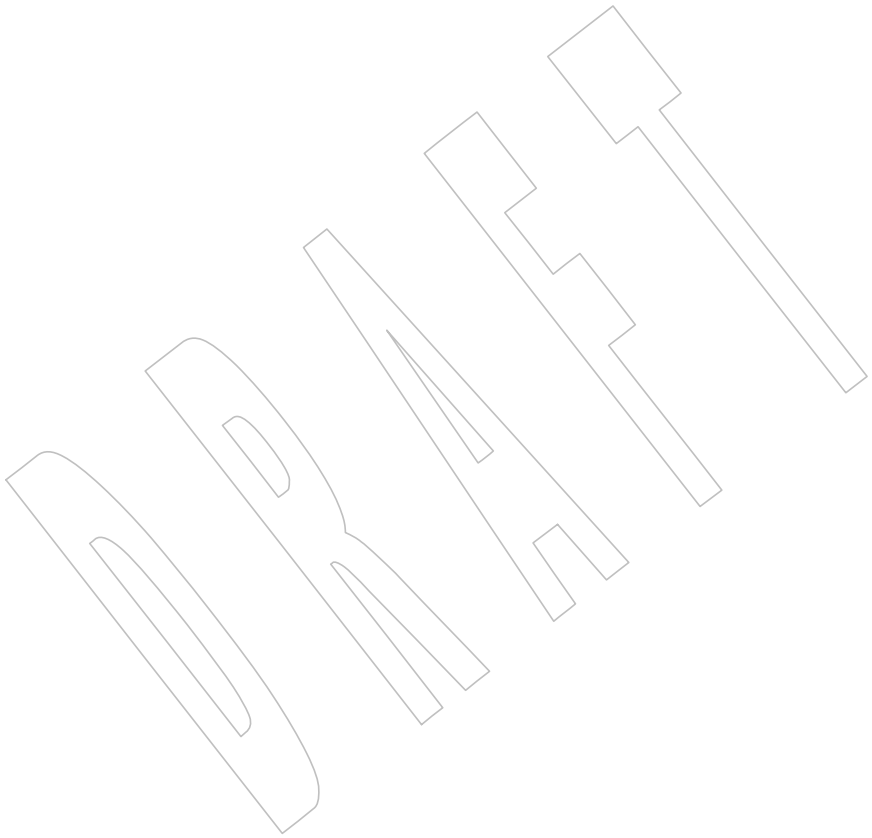
vsnprintf, vsprintf — format output of a stdarg argument list

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( )*.



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

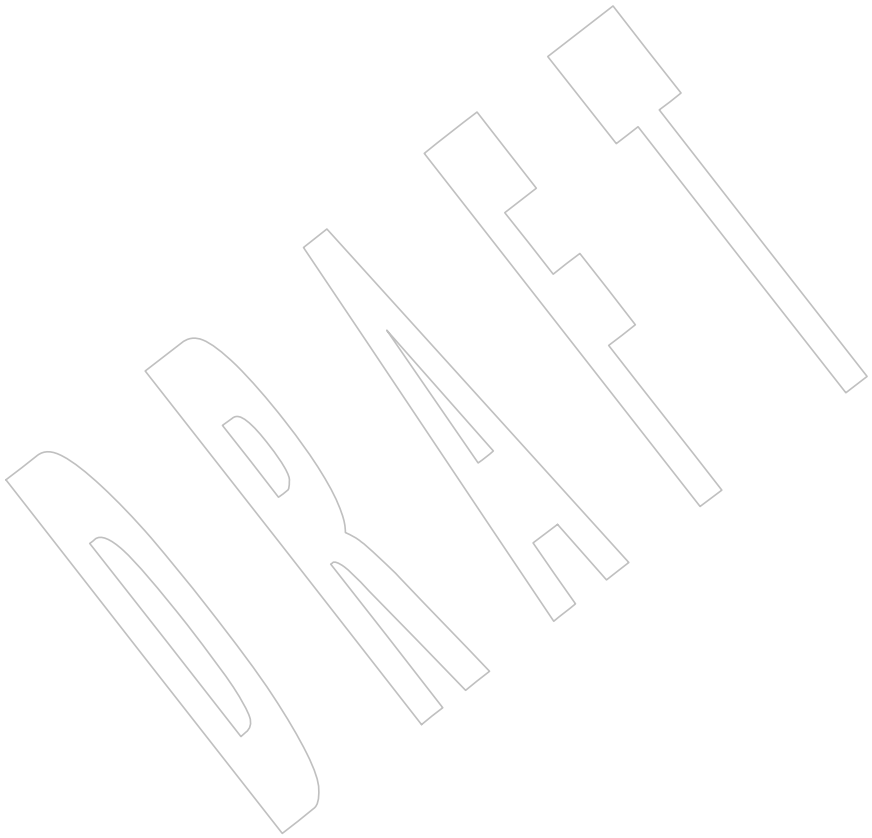
vswprintf — wide-character formatted output of a stdarg argument list

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 *vwscanf( )*.

51339 **NAME**

51340 vwprintf — wide-character formatted output of a stdarg argument list

51341 **SYNOPSIS**

51342 ```
#include <stdarg.h>
```

51343 ```
#include <stdio.h>
```

51344 ```
#include <wchar.h>
```

51345 ```
int vwprintf(const wchar_t *restrict format, va_list arg);
```

51346 **DESCRIPTION**

51347 Refer to *vfwprintf( )*.

51348 **NAME**

51349 vwscanf — wide-character formatted input of a stdarg argument list

51350 **SYNOPSIS**

51351
```
#include <stdarg.h>
```
51352
```
#include <stdio.h>
```
51353
```
#include <wchar.h>
```

51354
```
int vwscanf(const wchar_t *restrict format, va_list arg);
```

51355 **DESCRIPTION**

51356 Refer to *vwscanf( )*.

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:

XSI 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.

XSI If the calling process has SA_NOCLDWAIT set or has SIGCHLD set to SIG_IGN, and the process has no unwaited-for children that were transformed into zombie processes, the calling thread shall block until all of the children of the process containing the calling thread terminate, and

51401 *wait*( ) and *waitpid*( ) shall fail and set *errno* to [ECHILD].

51402 If *wait*( ) or *waitpid*( ) return because the status of a child process is available, these functions
51403 shall return a value equal to the process ID of the child process. In this case, if the value of the
51404 argument *stat_loc* is not a null pointer, information shall be stored in the location pointed to by
51405 *stat_loc*. The value stored at the location pointed to by *stat_loc* shall be 0 if and only if the status
51406 returned is from a terminated child process that terminated by one of the following means:

51407 1. The process returned 0 from *main*( ).

51408 2. The process called *_exit*( ) or *exit*( ) with a *status* argument of 0.

51409 3. The process was terminated because the last thread in the process terminated.

51410 Regardless of its value, this information may be interpreted using the following macros, which
51411 are defined in **<sys/wait.h>** and evaluate to integral expressions; the *stat_val* argument is the
51412 integer value pointed to by *stat_loc*.

51413 WIFEXITED(*stat_val*)
51414 Evaluates to a non-zero value if *status* was returned for a child process that terminated
51415 normally.

51416 WEXITSTATUS(*stat_val*)
51417 If the value of WIFEXITED(*stat_val*) is non-zero, this macro evaluates to the low-order 8 bits
51418 of the *status* argument that the child process passed to *_exit*( ) or *exit*( ), or the value the child
51419 process returned from *main*( ).

51420 WIFSIGNALED(*stat_val*)
51421 Evaluates to a non-zero value if *status* was returned for a child process that terminated due
51422 to the receipt of a signal that was not caught (see **<signal.h>**).

51423 WTERMSIG(*stat_val*)
51424 If the value of WIFSIGNALED(*stat_val*) is non-zero, this macro evaluates to the number of
51425 the signal that caused the termination of the child process.

51426 WIFSTOPPED(*stat_val*)
51427 Evaluates to a non-zero value if *status* was returned for a child process that is currently
51428 stopped.

51429 WSTOPSIG(*stat_val*)
51430 If the value of WIFSTOPPED(*stat_val*) is non-zero, this macro evaluates to the number of the
51431 signal that caused the child process to stop.

51432 XSI WIFCONTINUED(*stat_val*)
51433 Evaluates to a non-zero value if *status* was returned for a child process that has continued
51434 from a job control stop.

51435 SPN It is unspecified whether the *status* value returned by calls to *wait*( ) or *waitpid*( ) for processes
51436 created by *posix_spawn*( ) or *posix_spawnp*( ) can indicate a WIFSTOPPED(*stat_val*) before
51437 subsequent calls to *wait*( ) or *waitpid*( ) indicate WIFEXITED(*stat_val*) as the result of an error
51438 detected before the new process image starts executing.

51439 It is unspecified whether the *status* value returned by calls to *wait*( ) or *waitpid*( ) for processes
51440 created by *posix_spawn*( ) or *posix_spawnp*( ) can indicate a WIFSIGNALED(*stat_val*) if a signal is
51441 sent to the parent's process group after *posix_spawn*( ) or *posix_spawnp*( ) is called.

51442 If the information pointed to by *stat_loc* was stored by a call to *waitpid*( ) that specified the
51443 XSI WUNTRACED flag and did not specify the WCONTINUED flag, exactly one of the macros
51444 WIFEXITED(**stat_loc*), WIFSIGNALED(**stat_loc*), and WIFSTOPPED(**stat_loc*) shall evaluate to a
51445 non-zero value.

51446 If the information pointed to by *stat_loc* was stored by a call to *waitpid*( ) that specified the

XSI WUNTRACED and WCONTINUED flags, exactly one of the macros WIFEXITED(*stat_loc*), XSI 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 XSI WUNTRACED or WCONTINUED flags, or by a call to the *wait*( ) function, exactly one of the macros WIFEXITED(*stat_loc*) and WIFSIGNALED(*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 XSI WUNTRACED flag and specified the WCONTINUED flag, or by a call to the *wait*( ) function, XSI 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.

XSI 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.

[EINTR] The function was interrupted by a signal. The value of the location pointed to by *stat_loc* is undefined.

[EINVAL] The *options* argument is not valid.

EXAMPLES

Waiting for a Child Process and then Checking its Status

The following example demonstrates the use of *waitpid*(), *fork*(), and the macros used to interpret the status value returned by *waitpid*() (and *wait*()). The code segment creates a child process which does some unspecified work. Meanwhile the parent loops performing calls to *waitpid*() to monitor the status of the child. The loop terminates when child termination is detected.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
...

pid_t child_pid, wpid;
int status;

child_pid = fork();
if (child_pid == -1) {      /* fork() failed */
    perror("fork");
    exit(EXIT_FAILURE);
}

if (child_pid == 0) {       /* This is the child */
    /* Child does some work and then terminates */
    ...

} else {                    /* This is the parent */
    do {
        wpid = waitpid(child_pid, &status, WUNTRACED
#ifdef WCONTINUED       /* Not all implementations support this */
        | WCONTINUED
#endif
        );
        if (wpid == -1) {
            perror("waitpid");
            exit(EXIT_FAILURE);
        }

        if (WIFEXITED(status)) {
            printf("child exited, status=%d\n", WEXITSTATUS(status));

        } else if (WIFSIGNALED(status)) {
            printf("child killed (signal %d)\n", WTERMSIG(status));

        } else if (WIFSTOPPED(status)) {
            printf("child stopped (signal %d)\n", WSTOPSIG(status));

#ifdef WIFCONTINUED     /* Not all implementations support this */
        } else if (WIFCONTINUED(status)) {
            printf("child continued\n");
#endif
        } else {    /* Non-standard case -- may never happen */
            printf("Unexpected status (0x%x)\n", status);
        }
    } while (!WIFEXITED(status) && !WIFSIGNALED(status));
```



```
}
```

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 1003.1-200x 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 1003.1-200x, 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:

```
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 1003.1-200x 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 argument as **stat_loc* rather than *stat_val*. However, that did not follow the conventions of other specifications in this volume of IEEE Std 1003.1-200x or traditional usage. It also could have implied that the argument to the macro must literally be **stat_loc*; in fact, that value can be stored or passed as an argument to other functions before being interpreted by these macros.

The extension that affects *wait*( ) and *waitpid*( ) and is common in historical implementations is the *ptrace*( ) function. It is called by a child process and causes that child to stop and return a *status* that appears identical to the *status* indicated by WIFSTOPPED. The *status* of *ptrace*( )

children is traditionally returned regardless of the WUNTRACED flag (or by the *wait*( ) function). Most applications do not need to concern themselves with such extensions because they have control over what extensions they or their children use. However, applications, such as command interpreters, that invoke arbitrary processes may see this behavior when those arbitrary processes misuse such extensions.

Implementations that support **core** file creation or other implementation-defined actions on termination of some processes traditionally provide a bit in the *status* returned by *wait*( ) to indicate that such actions have occurred.

Allowing the *wait*( ) family of functions to discard a pending SIGCHLD signal that is associated with a successfully waited-for child process puts them into the *sigwait*( ) and *sigwaitinfo*( ) category with respect to SIGCHLD.

This definition allows implementations to treat a pending SIGCHLD signal as accepted by the process in *wait*( ), with the same meaning of "accepted" as when that word is applied to the *sigwait*( ) family of functions.

Allowing the *wait*( ) family of functions to behave this way permits an implementation to be able to deal precisely with SIGCHLD signals.

In particular, an implementation that does accept (discard) the SIGCHLD signal can make the following guarantees regardless of the queuing depth of signals in general (the list of waitable children can hold the SIGCHLD queue):

1. If a SIGCHLD signal handler is established via *sigaction*( ) without the SA_RESETHAND flag, SIGCHLD signals can be accurately counted; that is, exactly one SIGCHLD signal will be delivered to or accepted by the process for every child process that terminates.
2. A single *wait*( ) issued from a SIGCHLD signal handler can be guaranteed to return immediately with status information for a child process.
3. When SA_SIGINFO is requested, the SIGCHLD signal handler can be guaranteed to receive a non-NULL pointer to a **siginfo_t** structure that describes a child process for which a wait via *waitpid*( ) or *waitid*( ) will not block or fail.
4. The *system*( ) function will not cause the SIGCHLD handler of a process to be called as a result of the *fork*( )/*exec* executed within *system*( ) because *system*( ) will accept the SIGCHLD signal when it performs a *waitpid*( ) for its child process. This is a desirable behavior of *system*( ) so that it can be used in a library without causing side effects to the application linked with the library.

An implementation that does not permit the *wait*( ) family of functions to accept (discard) a pending SIGCHLD signal associated with a successfully waited-for child, cannot make the guarantees described above for the following reasons:

Guarantee #1
: Although it might be assumed that reliable queuing of all SIGCHLD signals generated by the system can make this guarantee, the counter-example is the case of a process that blocks SIGCHLD and performs an indefinite loop of *fork*( )/*wait*( ) operations. If the 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 1003.1-200x, Section 4.10, Memory Synchronization, **<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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/145 is applied, adding the example to the EXAMPLES section.

NAME

waitid — wait for a child process to change state

SYNOPSIS

```
#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*. The fields of the structure pointed to by *infop* are filled in as described for the SIGCHLD signal in **<signal.h>**. 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 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.

Applications shall specify at least one of the flags WEXITED, WSTOPPED, or WCONTINUED to be OR'd in with the *options* argument.

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 1003.1-200x, **<signal.h>**, **<sys/wait.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

The normative text is updated to avoid use of the term "must" for application requirements.

Issue 7

Austin Group Interpretation 1003.1-2001 #060 is applied, updating the DESCRIPTION.

The *waitid*( ) function is moved from the XSI option to the Base.

51728 **NAME**

51729 waitpid — wait for a child process to stop or terminate

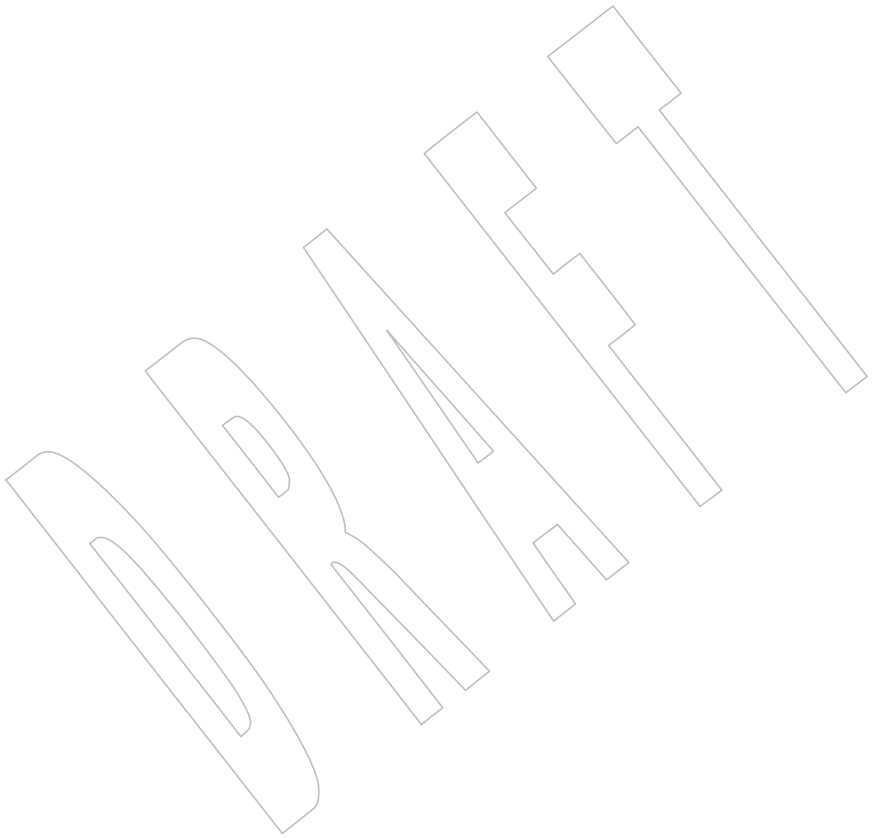
51730 **SYNOPSIS**

51731
```
#include <sys/wait.h>
```

51732
```
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

51733 **DESCRIPTION**

51734 Refer to *wait( )*.



51735 **NAME**

51736 wcpcpy — copy a wide-character string, returning a pointer to its end

51737 **SYNOPSIS**

51738 CX
```
#include <wchar.h>

wchar_t *wcpcpy(wchar_t *restrict ws1, const wchar_t *restrict ws2);
```
51739

51740 **DESCRIPTION**

51741 Refer to *wcscpy( )*.

51742 **NAME**

51743 wcpncpy — copy a fixed-size wide-character string, returning a pointer to its end

51744 **SYNOPSIS**

51745 CX
```
#include <wchar.h>

wchar_t *wcpncpy(wchar_t restrict *ws1, const wchar_t *restrict ws2,
    size_t n);
```

51748 **DESCRIPTION**

51749 Refer to *wcsncpy( )*.

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

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 1003.1-200x 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 1003.1-200x calls *wcrtomb*( ).

CX 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.

[EILSEQ] Invalid wide-character code is detected.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mbsinit( ), *wcsrtombs( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

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 normative text 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

wcscasecmp, wcscasecmp_l, wcsncasecmp, wcsncasecmp_l — case-insensitive wide-character string comparison

SYNOPSIS

CX
```
#include <wchar.h>

int wcscasecmp(const wchar_t *ws1, const wchar_t *ws2);
int wcscasecmp_l(const char *ws1, const char *ws2,
    locale_t locale);
int wcsncasecmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);
int wcsncasecmp_l(const char *ws1, const char *ws2,
    size_t n, locale_t locale);
```

DESCRIPTION

The *wcscasecmp*( ) and *wcsncasecmp*( ) functions are the wide-character equivalent of the *strcasecmp*( ) and *strncasecmp*( ) functions, respectively.

The *wcscasecmp*( ) and *wcscasecmp_l*( ) functions shall compare, while ignoring differences in case, the wide-character string pointed to by *ws1* to the wide-character string pointed to by *ws2*.

The *wcsncasecmp*( ) and *wcsncasecmp_l*( ) functions shall compare, while ignoring differences in case, not more than *n* wide-characters from the wide-character string pointed to by *ws1* to the wide-character string pointed to by *ws2*.

When the *LC_CTIME* category of the current locale is from the POSIX locale, these functions shall behave as if the strings had been converted to lowercase and then a byte comparison performed. Otherwise, the results are unspecified.

The information for *wcscasecmp_l*( ) and *wcsncasecmp_l*( ) about the case of the characters comes from the locale represented by *locale*.

RETURN VALUE

Upon completion, the *wcscasecmp*( ) and *wcscasecmp_l*( ) functions shall return an integer greater than, equal to, or less than 0 if the wide-character string pointed to by *ws1* is, ignoring case, greater than, equal to, or less than the wide-character string pointed to by *ws2*, respectively.

Upon completion, the *wcsncasecmp*( ) and *wcsncasecmp_l*( ) functions shall return an integer greater than, equal to, or less than 0 if the possibly null wide-character terminated string pointed to by *ws1* is, ignoring case, greater than, equal to, or less than the possibly null wide-character terminated string pointed to by *ws2*, respectively.

No return values are reserved to indicate an error.

ERRORS

The *wcscasecmp_l*( ) and *wcsncasecmp_l*( ) functions may fail if:

[EINVAL] *locale* is not a valid locale object handle.

51840 **EXAMPLES**

51841 None.

51842 **APPLICATION USAGE**

51843 None.

51844 **RATIONALE**

51845 None.

51846 **FUTURE DIRECTIONS**

51847 None.

51848 **SEE ALSO**

51849 *strcasecmp( )*, *wcscmp( )*, *wcsncmp( )*, the Base Definitions volume of IEEE Std 1003.1-200x,
51850 **<wchar.h>**

51851 **CHANGE HISTORY**

51852 First released in Issue 7.

NAME

wcscat — concatenate two wide-character strings

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcscat(wchar_t *restrict ws1, const wchar_t *restrict ws2);
```

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 1003.1-200x 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 1003.1-200x, **<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.

NAME

wcschr — wide-character string scanning operation

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcschr(const wchar_t *ws, wchar_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 1003.1-200x defers to the ISO C standard.

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.

RETURN VALUE

Upon completion, *wcschr*( ) shall return a pointer to the wide-character code, or a null pointer if the wide-character code is not found.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcsrchr( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 6

The normative text 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

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 1003.1-200x defers to the ISO C standard.

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

wcscasecmp( ), *wcsncmp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

NAME

wcscoll, wcscoll_l — wide-character string comparison using collating information

SYNOPSIS

```
#include <wchar.h>

int wcscoll(const wchar_t *ws1, const wchar_t *ws2);
CX int wcscoll_l(const wchar_t *ws1, const wchar_t *ws2,
       locale_t locale);
```

DESCRIPTION

CX For *wcscoll*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

CX The *wcscoll*( ) and *wcscoll_l*( ) functions 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 CX *LC_COLLATE* category of the current locale of the process, or the locale represented by *locale*, respectively.

CX The *wcscoll*( ) and *wcscoll_l*( ) functions shall not change the setting of *errno* if successful.

CX An application wishing to check for error situations should set *errno* to 0 before calling *wcscoll*( ) or *wcscoll_l*( ). If *errno* is non-zero on return, an error has occurred.

RETURN VALUE

CX Upon successful completion, *wcscoll*( ) and *wcscoll_l*( ) 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 CX interpreted as appropriate to the current locale, or to the locale represented by *locale*, CX respectively. On error, *wcscoll*( ) and *wcscoll_l*( ) shall set *errno*, but no return value is reserved to indicate an error.

ERRORS

These functions may fail if:

CX [EINVAL] The *ws1* or *ws2* arguments contain wide-character codes outside the domain of the collating sequence.

The *wcscoll_l*( ) function may fail if:

CX [EINVAL] *locale* is not a valid locale object handle.

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 1003.1-200x, **<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 7

The *wcscoll_l( )* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.

NAME

wcpcpy, wcscpy — copy a wide-character string, returning a pointer to its end

SYNOPSIS

```
#include <wchar.h>

CX  wchar_t *wcpcpy(wchar_t *restrict ws1, const wchar_t *restrict ws2);
    wchar_t *wcscpy(wchar_t *restrict ws1, const wchar_t *restrict ws2);
```

DESCRIPTION

CX For *wcscpy*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

CX The *wcpcpy*( ) and *wcscpy*( ) functions shall copy the wide-character string pointed to by *ws2* (including the terminating null wide-character code) into the array pointed to by *ws1*.

The application shall ensure that there is room for at least *wcslen*(*ws2*)+1 wide characters in the *ws1* array, and that the *ws2* and *ws1* arrays do not overlap.

If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE

CX The *wcpcpy*( ) function shall return a pointer to the terminating null wide-character code copied into the *ws1* buffer.

The *wcscpy*( ) function shall return *ws1*.

No return values are reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strcpy( ), *wcsdup*( ), *wcsncpy*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<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.

Issue 7

The *wcpcpy*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 1.

52038 **NAME**

52039 wcscspn — get the length of a complementary wide substring

52040 **SYNOPSIS**

52041
```
#include <wchar.h>
```

52042
```
size_t wcscspn(const wchar_t *ws1, const wchar_t *ws2);
```

52043 **DESCRIPTION**

52044 CX The functionality described on this reference page is aligned with the ISO C standard. Any
52045 conflict between the requirements described here and the ISO C standard is unintentional. This
52046 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

52047 The *wcscspn*( ) function shall compute the length (in wide characters) of the maximum initial
52048 segment of the wide-character string pointed to by *ws1* which consists entirely of wide-character
52049 codes *not* from the wide-character string pointed to by *ws2*.

52050 **RETURN VALUE**

52051 The *wcscspn*( ) function shall return the length of the initial substring of *ws1*; no return value is
52052 reserved to indicate an error.

52053 **ERRORS**

52054 No errors are defined.

52055 **EXAMPLES**

52056 None.

52057 **APPLICATION USAGE**

52058 None.

52059 **RATIONALE**

52060 None.

52061 **FUTURE DIRECTIONS**

52062 None.

52063 **SEE ALSO**

52064 *wcsspn*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

52065 **CHANGE HISTORY**

52066 First released in Issue 4. Derived from the MSE working draft.

52067 **Issue 5**

52068 The RETURN VALUE section is updated to indicate that *wcscspn*( ) returns the length of *ws1*,
52069 rather than *ws1* itself.

52070 **NAME**

52071 wcsdup — duplicate a wide-character string

52072 **SYNOPSIS**

52073 CX
```
#include <wchar.h>

wchar_t *wcsdup(const wchar_t *string);
```

52075 **DESCRIPTION**

52076 The *wcsdup*( ) function is the wide-character equivalent of the *strdup*( ) function.

52077 The *wcsdup*( ) function shall return a pointer to a new wide-character string, which is the
52078 duplicate of the wide-character string *string*. The returned pointer can be passed to *free*( ). A
52079 null pointer is returned if the new wide-character string cannot be created.

52080 **RETURN VALUE**

52081 Upon successful completion, the *wcsdup*( ) function shall return a pointer to the newly allocated
52082 wide-character string. Otherwise, it shall return a null pointer and set *errno* to indicate the error.

52083 **ERRORS**

52084 The *wcsdup*( ) function shall fail if:

52085 [ENOMEM] Memory large enough for the duplicate string could not be allocated.

52086 **EXAMPLES**

52087 None.

52088 **APPLICATION USAGE**

52089 None.

52090 **RATIONALE**

52091 None.

52092 **FUTURE DIRECTIONS**

52093 None.

52094 **SEE ALSO**

52095 *free*( ), *strdup*( ), *wcscpy*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

52096 **CHANGE HISTORY**

52097 First released in Issue 7.

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

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 1003.1-200x defers to the ISO C standard.

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 1003.1-200x, **<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** *.

Issue 6

The *wcsftime*( ) prototype is updated for alignment with the ISO/IEC 9899: 1999 standard.

52142 **NAME**

52143 wcslen, wcsnlen — get length of a fixed-sized wide-character string

52144 **SYNOPSIS**

52145
```
#include <wchar.h>
```

52146
```
size_t wcslen(const wchar_t *ws);
```

52147 CX
```
size_t wcsnlen(const wchar_t *ws, size_t maxlen);
```

52148 **DESCRIPTION**

52149 CX For *wcslen*( ): The functionality described on this reference page is aligned with the ISO C
52150 standard. Any conflict between the requirements described here and the ISO C standard is
52151 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

52152 The *wcslen*( ) function shall compute the number of wide-character codes in the wide-character
52153 string to which *ws* points, not including the terminating null wide-character code.

52154 CX The *wcsnlen*( ) function shall compute the smaller of the number of wide characters in the string
52155 to which *ws* points, not including the terminating null wide-character code, and the value of
52156 *maxlen*. The *wcsnlen*( ) function shall never examine more than the first *maxlen* characters of the
52157 wide-character string pointed to by *ws*.

52158 **RETURN VALUE**

52159 The *wcslen*( ) function shall return the length of *ws*.

52160 CX The *wcsnlen*( ) function shall return an integer containing the smaller of either the length of the
52161 wide-character string pointed to by *ws* or *maxlen*.

52162 No return values are reserved to indicate an error.

52163 **ERRORS**

52164 No errors are defined.

52165 **EXAMPLES**

52166 None.

52167 **APPLICATION USAGE**

52168 None.

52169 **RATIONALE**

52170 None.

52171 **FUTURE DIRECTIONS**

52172 None.

52173 **SEE ALSO**

52174 *strlen*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

52175 **CHANGE HISTORY**

52176 First released in Issue 4. Derived from the MSE working draft.

52177 **Issue 7**

52178 The *wcsnlen*( ) function is added from The Open Group Technical Standard, 2006, Extended API
52179 Set Part 1.

52180 **NAME**

52181 wcsncasecmp, wcsncasecmp_l — case-insensitive wide-character string comparison

52182 **SYNOPSIS**

52183 CX
```
#include <wchar.h>

int wcsncasecmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);
int wcsncasecmp_l(const char *ws1, const char *ws2,
    size_t n, locale_t locale);
```

52187 **DESCRIPTION**

52188 Refer to *wcscasecmp( )*.

52189 **NAME**

52190 wcsncat — concatenate a wide-character string with part of another

52191 **SYNOPSIS**

52192
```
#include <wchar.h>
```

52193
```
wchar_t *wcsncat(wchar_t *restrict ws1, const wchar_t *restrict ws2,
    size_t n);
```

52195 **DESCRIPTION**

52196 CX The functionality described on this reference page is aligned with the ISO C standard. Any
52197 conflict between the requirements described here and the ISO C standard is unintentional. This
52198 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

52199 The *wcsncat*( ) function shall append not more than *n* wide-character codes (a null wide-
52200 character code and wide-character codes that follow it are not appended) from the array pointed
52201 to by *ws2* to the end of the wide-character string pointed to by *ws1*. The initial wide-character
52202 code of *ws2* shall overwrite the null wide-character code at the end of *ws1*. A terminating null
52203 wide-character code shall always be appended to the result. If copying takes place between
52204 objects that overlap, the behavior is undefined.

52205 **RETURN VALUE**

52206 The *wcsncat*( ) function shall return *ws1*; no return value is reserved to indicate an error.

52207 **ERRORS**

52208 No errors are defined.

52209 **EXAMPLES**

52210 None.

52211 **APPLICATION USAGE**

52212 None.

52213 **RATIONALE**

52214 None.

52215 **FUTURE DIRECTIONS**

52216 None.

52217 **SEE ALSO**

52218 *wcscat*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

52219 **CHANGE HISTORY**

52220 First released in Issue 4. Derived from the MSE working draft.

52221 **Issue 6**

52222 The *wcsncat*( ) prototype is updated for alignment with the ISO/IEC 9899: 1999 standard.

52223 **NAME**

52224 wcsncmp — compare part of two wide-character strings

52225 **SYNOPSIS**

52226
```
#include <wchar.h>
```

52227
```
int wcsncmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);
```

52228 **DESCRIPTION**

52229 CX The functionality described on this reference page is aligned with the ISO C standard. Any
52230 conflict between the requirements described here and the ISO C standard is unintentional. This
52231 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

52232 The *wcsncmp*( ) function shall compare not more than *n* wide-character codes (wide-character
52233 codes that follow a null wide-character code are not compared) from the array pointed to by *ws1*
52234 to the array pointed to by *ws2*.

52235 The sign of a non-zero return value shall be determined by the sign of the difference between the
52236 values of the first pair of wide-character codes that differ in the objects being compared.

52237 **RETURN VALUE**

52238 Upon successful completion, *wcsncmp*( ) shall return an integer greater than, equal to, or less
52239 than 0, if the possibly null-terminated array pointed to by *ws1* is greater than, equal to, or less
52240 than the possibly null-terminated array pointed to by *ws2*, respectively.

52241 **ERRORS**

52242 No errors are defined.

52243 **EXAMPLES**

52244 None.

52245 **APPLICATION USAGE**

52246 None.

52247 **RATIONALE**

52248 None.

52249 **FUTURE DIRECTIONS**

52250 None.

52251 **SEE ALSO**

52252 *wcscasecmp*( ), *wcscmp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

52253 **CHANGE HISTORY**

52254 First released in Issue 4. Derived from the MSE working draft.

52255 **NAME**

52256 wcpncpy, wcsncpy — copy a fixed-size wide-character string, returning a pointer to its end

52257 **SYNOPSIS**

52258
```
#include <wchar.h>
```

52259 CX
```
wchar_t *wcpncpy(wchar_t restrict *ws1, const wchar_t *restrict ws2,
    size_t n);
```
52260
52261
```
wchar_t *wcsncpy(wchar_t *restrict ws1, const wchar_t *restrict ws2,
    size_t n);
```
52262

52263 **DESCRIPTION**

52264 CX For *wcsncpy*( ): The functionality described on this reference page is aligned with the ISO C
52265 standard. Any conflict between the requirements described here and the ISO C standard is
52266 unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

52267 CX The *wcpncpy*( ) and *wcsncpy*( ) functions shall copy not more than *n* wide-character codes (wide-
52268 character codes that follow a null wide-character code are not copied) from the array pointed to
52269 by *ws2* to the array pointed to by *ws1*. If copying takes place between objects that overlap, the
52270 behavior is undefined.

52271 If the array pointed to by *ws2* is a wide-character string that is shorter than *n* wide-character
52272 codes, null wide-character codes shall be appended to the copy in the array pointed to by *ws1*,
52273 until *n* wide-character codes in all are written.

52274 **RETURN VALUE**

52275 CX If any null wide-character codes were written into the destination, the *wcpncpy*( ) function shall
52276 return the address of the first such null wide-character code. Otherwise, it shall return *&ws1*[*n*].

52277 The *wcsncpy*( ) function shall return *ws1*.

52278 No return values are reserved to indicate an error.

52279 **ERRORS**

52280 No errors are defined.

52281 **EXAMPLES**

52282 None.

52283 **APPLICATION USAGE**

52284 If there is no null wide-character code in the first *n* wide-character codes of the array pointed to
52285 by *ws2*, the result is not null-terminated.

52286 **RATIONALE**

52287 None.

52288 **FUTURE DIRECTIONS**

52289 None.

52290 **SEE ALSO**

52291 *strncpy*( ), *wcscpy*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

52292 **CHANGE HISTORY**

52293 First released in Issue 4. Derived from the MSE working draft.

52294 **Issue 6**

52295 The *wcsncpy*( ) prototype is updated for alignment with the ISO/IEC 9899: 1999 standard.

Issue 7

The *wcpncpy*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 1.

52299 **NAME**

52300 wcsnlen — get length of a fixed-sized wide-character string

52301 **SYNOPSIS**

52302 CX
```
#include <wchar.h>

size_t wcsnlen(const wchar_t *ws, size_t maxlen);
```

52304 **DESCRIPTION**

52305 Refer to *wcslen( )*.

52306 **NAME**

52307 wcsnrtombs — convert wide-character string to multi-byte string

52308 **SYNOPSIS**

52309 CX
```
#include <wchar.h>

size_t wcsnrtombs(char *dst, const wchar_t **src, size_t nwc,
    size_t len, mbstate_t *ps);
```

52312 **DESCRIPTION**

52313 Refer to *wcsrtombs( )*.

52314 **NAME**

52315 wcspbrk — scan a wide-character string for a wide-character code

52316 **SYNOPSIS**

52317
```
#include <wchar.h>
```

52318
```
wchar_t *wcspbrk(const wchar_t *ws1, const wchar_t *ws2);
```

52319 **DESCRIPTION**

52320 CX The functionality described on this reference page is aligned with the ISO C standard. Any
52321 conflict between the requirements described here and the ISO C standard is unintentional. This
52322 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

52323 The *wcspbrk*( ) function shall locate the first occurrence in the wide-character string pointed to by
52324 *ws1* of any wide-character code from the wide-character string pointed to by *ws2*.

52325 **RETURN VALUE**

52326 Upon successful completion, *wcspbrk*( ) shall return a pointer to the wide-character code or a null
52327 pointer if no wide-character code from *ws2* occurs in *ws1*.

52328 **ERRORS**

52329 No errors are defined.

52330 **EXAMPLES**

52331 None.

52332 **APPLICATION USAGE**

52333 None.

52334 **RATIONALE**

52335 None.

52336 **FUTURE DIRECTIONS**

52337 None.

52338 **SEE ALSO**

52339 *wcschr*( ), *wcsrchr*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

52340 **CHANGE HISTORY**

52341 First released in Issue 4. Derived from the MSE working draft.

52342 **NAME**

52343 wcsrchr — wide-character string scanning operation

52344 **SYNOPSIS**

52345
```
#include <wchar.h>
```

52346
```
wchar_t *wcsrchr(const wchar_t *ws, wchar_t wc);
```

52347 **DESCRIPTION**

52348 CX The functionality described on this reference page is aligned with the ISO C standard. Any
52349 conflict between the requirements described here and the ISO C standard is unintentional. This
52350 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

52351 The *wcsrchr*( ) function shall locate the last occurrence of *wc* in the wide-character string pointed
52352 to by *ws*. The application shall ensure that the value of *wc* is a character representable as a type
52353 **wchar_t** and a wide-character code corresponding to a valid character in the current locale. The
52354 terminating null wide-character code shall be considered to be part of the wide-character string.

52355 **RETURN VALUE**

52356 Upon successful completion, *wcsrchr*( ) shall return a pointer to the wide-character code or a null
52357 pointer if *wc* does not occur in the wide-character string.

52358 **ERRORS**

52359 No errors are defined.

52360 **EXAMPLES**

52361 None.

52362 **APPLICATION USAGE**

52363 None.

52364 **RATIONALE**

52365 None.

52366 **FUTURE DIRECTIONS**

52367 None.

52368 **SEE ALSO**

52369 *wcschr*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

52370 **CHANGE HISTORY**

52371 First released in Issue 4. Derived from the MSE working draft.

52372 **Issue 6**

52373 The normative text is updated to avoid use of the term "must" for application requirements.

NAME

wcsnrtombs, wcsrtombs — convert a wide-character string to a character string (restartable)

SYNOPSIS

```
#include <wchar.h>

CX  size_t wcsnrtombs(char *dst, const wchar_t **src, size_t nwc,
        size_t len, mbstate_t *ps);
    size_t wcsrtombs(char *restrict dst, const wchar_t **restrict src,
        size_t len, mbstate_t *restrict ps);
```

DESCRIPTION

CX For *wcsrtombs*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x 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.

CX If the application uses any of the _POSIX_THREAD_SAFE_FUNCTIONS or _POSIX_THREADS functions, the application shall ensure that the *wcsrtombs*( ) function is called with a non-NULL *ps* argument.

The *wcsnrtombs*( ) function shall be equivalent to the *wcsrtombs*( ) function, except that the conversion is limited to the first *nwc* wide characters.

The behavior of these functions shall be affected by the *LC_CTYPE* category of the current locale.

The implementation shall behave as if no function defined in System Interfaces volume of IEEE Std 1003.1-200x calls these functions.

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, these functions shall store the value of the macro [EILSEQ] in *errno* and return (**size_t**)–1; the conversion state is undefined. Otherwise, these functions shall return the number of bytes in the resulting character sequence, not including the terminating null (if any).

52418 **ERRORS**

52419 These functions may fail if:

52420 CX [EINVAL] *ps* points to an object that contains an invalid conversion state.

52421 [EILSEQ] A wide-character code does not correspond to a valid character.

52422 **EXAMPLES**

52423 None.

52424 **APPLICATION USAGE**

52425 None.

52426 **RATIONALE**

52427 None.

52428 **FUTURE DIRECTIONS**

52429 None.

52430 **SEE ALSO**

52431 *mbsinit( )*, *wcrtomb( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

52432 **CHANGE HISTORY**

52433 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
52434 (E).

52435 **Issue 6**

52436 In the DESCRIPTION, a note on using this function in a threaded application is added.

52437 Extensions beyond the ISO C standard are marked.

52438 The normative text is updated to avoid use of the term "must" for application requirements.

52439 The *wcsrtombs( )* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

52440 **Issue 7**

52441 The *wcnsrtombs( )* function is added from The Open Group Technical Standard, 2006, Extended
52442 API Set Part 1.

52443 **NAME**

52444 wcsspn — get the length of a wide substring

52445 **SYNOPSIS**

52446
```
#include <wchar.h>
```

52447
```
size_t wcsspn(const wchar_t *ws1, const wchar_t *ws2);
```

52448 **DESCRIPTION**

52449 CX The functionality described on this reference page is aligned with the ISO C standard. Any
52450 conflict between the requirements described here and the ISO C standard is unintentional. This
52451 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

52452 The *wcsspn*( ) function shall compute the length (in wide characters) of the maximum initial
52453 segment of the wide-character string pointed to by *ws1* which consists entirely of wide-character
52454 codes from the wide-character string pointed to by *ws2*.

52455 **RETURN VALUE**

52456 The *wcsspn*( ) function shall return the length of the initial substring of *ws1*; no return value is
52457 reserved to indicate an error.

52458 **ERRORS**

52459 No errors are defined.

52460 **EXAMPLES**

52461 None.

52462 **APPLICATION USAGE**

52463 None.

52464 **RATIONALE**

52465 None.

52466 **FUTURE DIRECTIONS**

52467 None.

52468 **SEE ALSO**

52469 *wcscspn*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

52470 **CHANGE HISTORY**

52471 First released in Issue 4. Derived from the MSE working draft.

52472 **Issue 5**

52473 The RETURN VALUE section is updated to indicate that *wcsspn*( ) returns the length of *ws1*
52474 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

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 1003.1-200x 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 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899: 1990/Amendment 1: 1995 (E).

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

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 1003.1-200x defers to the ISO C standard.

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 consisting of the wide character `'e'` or the wide character `'E'`, optionally followed by a `'+'` or `'−'` wide character, and then followed by one or more decimal digits
- A 0x or 0X, then a non-empty sequence of hexadecimal digits optionally containing a radix character; then an optional binary exponent part consisting of the wide character `'p'` or the wide character `'P'`, optionally folowed by a `'+'` or `'−'` wide character, and then followed by one or more decimal digits
- One of INF or INFINITY, or any other wide string equivalent except for case
- One of NAN or NAN(*n-wchar-sequence*$_{opt}$), 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-sequence*$_{opt}$) shall be interpreted as a quiet NaN, if supported in the return type, else as if it were a subject sequence part that does not have the expected form; the meaning of the n-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.

CX The radix character shall be as defined in the locale of the process (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 (`'.'`).

CX 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.

CX 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
CX could be performed, 0 shall be returned and *errno* may be set to [EINVAL].

If the correct value is outside the range of representable values, ±HUGE_VAL, ±HUGE_VALF, or ±HUGE_VALL shall be returned (according to the sign of the value), and *errno* shall be set to [ERANGE].

If the correct value would cause 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 1003.1-200x, 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.

ISO/IEC 9899: 1999 standard, Technical Corrigendum 1 is incorporated.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/66 is applied, correcting the second paragraph in the RETURN VALUE section.

52637 **Issue 7**

52638 Austin Group Interpretation 1003.1-2001 #015 is applied.

52639 **NAME**

52640 wcstoimax, wcstoumax — convert a wide-character string to an integer type

52641 **SYNOPSIS**

52642 ```
#include <stddef.h>
```

52643 ```
#include <inttypes.h>
```

52644 ```
intmax_t wcstoimax(const wchar_t *restrict nptr,
```

52645 ```
    wchar_t **restrict endptr, int base);
```

52646 ```
uintmax_t wcstoumax(const wchar_t *restrict nptr,
```

52647 ```
    wchar_t **restrict endptr, int base);
```

52648 **DESCRIPTION**

52649 CX The functionality described on this reference page is aligned with the ISO C standard. Any
52650 conflict between the requirements described here and the ISO C standard is unintentional. This
52651 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

52652 These functions shall be equivalent to the *wcstol*( ), *wcstoll*( ), *wcstoul*( ), and *wcstoull*( ) functions,
52653 respectively, except that the initial portion of the wide string shall be converted to **intmax_t** and
52654 **uintmax_t** representation, respectively.

52655 **RETURN VALUE**

52656 These functions shall return the converted value, if any.

52657 If no conversion could be performed, zero shall be returned. If the correct value is outside the
52658 range of representable values, {INTMAX_MAX}, {INTMAX_MIN}, or {UINTMAX_MAX} shall
52659 be returned (according to the return type and sign of the value, if any), and *errno* shall be set to
52660 [ERANGE].

52661 **ERRORS**

52662 These functions shall fail if:

52663 [EINVAL] The value of *base* is not supported.

52664 [ERANGE] The value to be returned is not representable.

52665 These functions may fail if:

52666 [EINVAL] No conversion could be performed.

52667 **EXAMPLES**

52668 None.

52669 **APPLICATION USAGE**

52670 None.

52671 **RATIONALE**

52672 None.

52673 **FUTURE DIRECTIONS**

52674 None.

52675 **SEE ALSO**

52676 *wcstol*( ), *wcstoul*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<inttypes.h>**,
52677 **<stddef.h>**

52678 **CHANGE HISTORY**
52679 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

52680 **NAME**

52681 wcstok — split a wide-character string into tokens

52682 **SYNOPSIS**

52683
```
#include <wchar.h>
```

52684
```
wchar_t *wcstok(wchar_t *restrict ws1, const wchar_t *restrict ws2,
    wchar_t **restrict ptr);
```

52686 **DESCRIPTION**

52687 CX The functionality described on this reference page is aligned with the ISO C standard. Any
52688 conflict between the requirements described here and the ISO C standard is unintentional. This
52689 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

52690 A sequence of calls to *wcstok*( ) shall break the wide-character string pointed to by *ws1* into a
52691 sequence of tokens, each of which shall be delimited by a wide-character code from the wide-
52692 character string pointed to by *ws2*. The *ptr* argument points to a caller-provided **wchar_t** pointer
52693 into which the *wcstok*( ) function shall store information necessary for it to continue scanning the
52694 same wide-character string.

52695 The first call in the sequence has *ws1* as its first argument, and is followed by calls with a null
52696 pointer as their first argument. The separator string pointed to by *ws2* may be different from call
52697 to call.

52698 The first call in the sequence shall search the wide-character string pointed to by *ws1* for the first
52699 wide-character code that is *not* contained in the current separator string pointed to by *ws2*. If no
52700 such wide-character code is found, then there are no tokens in the wide-character string pointed
52701 to by *ws1* and *wcstok*( ) shall return a null pointer. If such a wide-character code is found, it shall
52702 be the start of the first token.

52703 The *wcstok*( ) function shall then search from there for a wide-character code that *is* contained in
52704 the current separator string. If no such wide-character code is found, the current token extends
52705 to the end of the wide-character string pointed to by *ws1*, and subsequent searches for a token
52706 shall return a null pointer. If such a wide-character code is found, it shall be overwritten by a
52707 null wide character, which terminates the current token. The *wcstok*( ) function shall save a
52708 pointer to the following wide-character code, from which the next search for a token shall start.

52709 Each subsequent call, with a null pointer as the value of the first argument, shall start searching
52710 from the saved pointer and behave as described above.

52711 The implementation shall behave as if no function calls *wcstok*( ).

52712 **RETURN VALUE**

52713 Upon successful completion, the *wcstok*( ) function shall return a pointer to the first wide-
52714 character code of a token. Otherwise, if there is no token, *wcstok*( ) shall return a null pointer.

52715 **ERRORS**

52716 No errors are defined.

52717 **EXAMPLES**

52718 None.

52719 **APPLICATION USAGE**

52720 None.

52721 **RATIONALE**

52722 None.

52723 **FUTURE DIRECTIONS**

52724 None.

52725 **SEE ALSO**

52726 The Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

52727 **CHANGE HISTORY**

52728 First released in Issue 4.

52729 **Issue 5**

52730 Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, a third argument is
52731 added to the definition of *wcstok*( ) in the SYNOPSIS.

52732 **Issue 6**

52733 The *wcstok*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

52734 **NAME**

52735 wcstol, wcstoll — convert a wide-character string to a long integer

52736 **SYNOPSIS**

52737
```
#include <wchar.h>
```

52738
```
long wcstol(const wchar_t *restrict nptr, wchar_t **restrict endptr,
    int base);
```
52739
52740
```
long long wcstoll(const wchar_t *restrict nptr,
    wchar_t **restrict endptr, int base);
```
52741

52742 **DESCRIPTION**

52743 CX The functionality described on this reference page is aligned with the ISO C standard. Any
52744 conflict between the requirements described here and the ISO C standard is unintentional. This
52745 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

52746 These functions shall convert the initial portion of the wide-character string pointed to by *nptr* to
52747 **long** and **long long**, respectively. First, they shall decompose the input string into three parts:

52748 1. An initial, possibly empty, sequence of white-space wide-character codes (as specified by
52749 *iswspace*( ))

52750 2. A subject sequence interpreted as an integer represented in some radix determined by the
52751 value of *base*

52752 3. A final wide-character string of one or more unrecognized wide-character codes,
52753 including the terminating null wide-character code of the input wide-character string

52754 Then they shall attempt to convert the subject sequence to an integer, and return the result.

52755 If *base* is 0, the expected form of the subject sequence is that of a decimal constant, octal constant,
52756 or hexadecimal constant, any of which may be preceded by a `'+'` or `'−'` sign. A decimal
52757 constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal
52758 constant consists of the prefix `'0'` optionally followed by a sequence of the digits `'0'` to `'7'`
52759 only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the
52760 decimal digits and letters `'a'` (or `'A'`) to `'f'` (or `'F'`) with values 10 to 15 respectively.

52761 If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence
52762 of letters and digits representing an integer with the radix specified by *base*, optionally preceded
52763 by a `'+'` or `'−'` sign, but not including an integer suffix. The letters from `'a'` (or `'A'`) to `'z'`
52764 (or `'Z'`) inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less
52765 than that of *base* shall be permitted. If the value of *base* is 16, the wide-character code
52766 representations of 0x or 0X may optionally precede the sequence of letters and digits, following
52767 the sign if present.

52768 The subject sequence is defined as the longest initial subsequence of the input wide-character
52769 string, starting with the first non-white-space wide-character code that is of the expected form.
52770 The subject sequence contains no wide-character codes if the input wide-character string is
52771 empty or consists entirely of white-space wide-character code, or if the first non-white-space
52772 wide-character code is other than a sign or a permissible letter or digit.

52773 If the subject sequence has the expected form and *base* is 0, the sequence of wide-character codes
52774 starting with the first digit shall be interpreted as an integer constant. If the subject sequence has
52775 the expected form and the value of *base* is between 2 and 36, it shall be used as the base for
52776 conversion, ascribing to each letter its value as given above. If the subject sequence begins with a
52777 minus sign, the value resulting from the conversion shall be negated. A pointer to the final wide-
52778 character string shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a
52779 null pointer.

CX 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.

CX 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 CX 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:

CX [EINVAL] The value of *base* is not supported.

[ERANGE] The value to be returned is not representable.

These functions may fail if:

CX [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 1003.1-200x, **<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.

Issue 7

SD5-XSH-ERN-56 is applied, removing the reference to **unsigned long** and **unsigned long long** from the DESCRIPTION.

52827 **NAME**

52828 wcstold — convert a wide-character string to a double-precision number

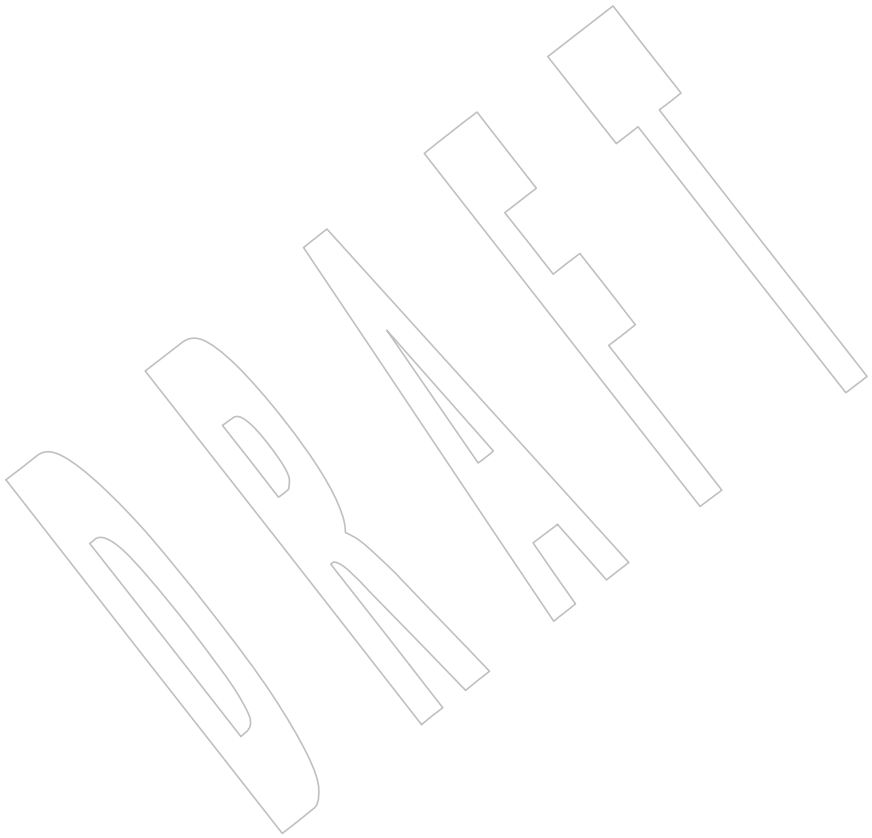
52829 **SYNOPSIS**

52830
```
#include <wchar.h>
```

52831
```
long double wcstold(const wchar_t *restrict nptr,
```
52832
```
    wchar_t **restrict endptr);
```

52833 **DESCRIPTION**

52834 Refer to *wcstod( )*.



52835 **NAME**

52836 wcstoll — convert a wide-character string to a long integer

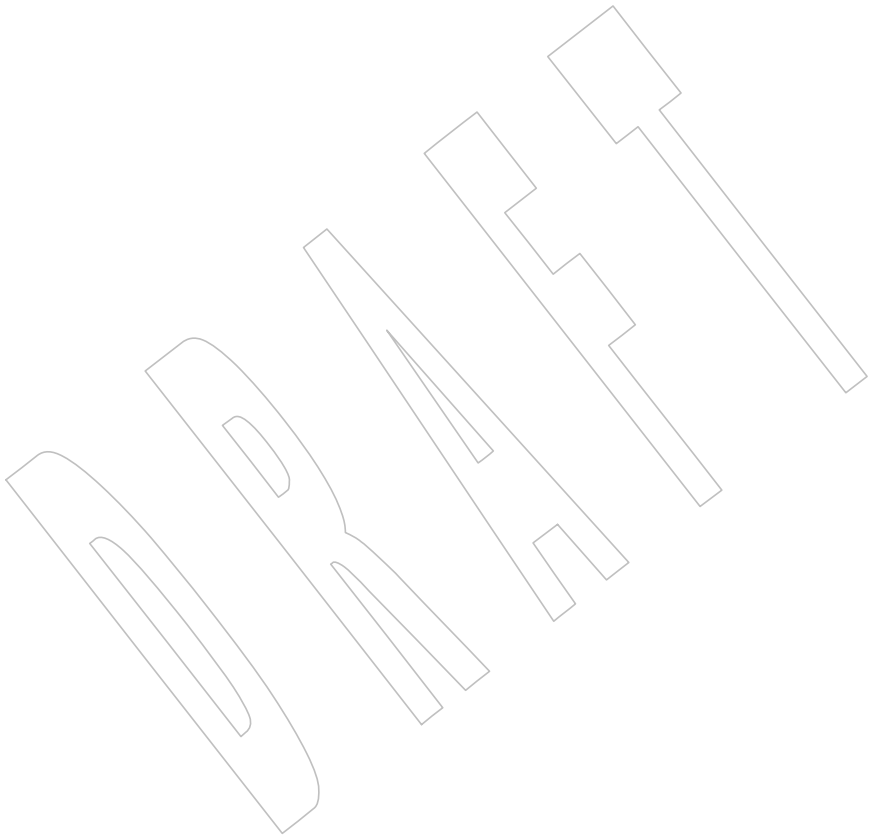
52837 **SYNOPSIS**

52838
```
#include <wchar.h>
```

52839
```
long long wcstoll(const wchar_t *restrict nptr,
```
52840
```
    wchar_t **restrict endptr, int base);
```

52841 **DESCRIPTION**

52842 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

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 1003.1-200x defers to the ISO C standard.

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 CX 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 thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

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:

CX [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 1003.1-200x, **<stdlib.h>**

52884 **CHANGE HISTORY**

52885 First released in Issue 4. Derived from the ISO C standard.

52886 **Issue 6**

52887 The following new requirements on POSIX implementations derive from alignment with the
52888 Single UNIX Specification:

52889 • The DESCRIPTION states the effect of when *s* is a null pointer.

52890 • The [EILSEQ] error condition is added.

52891 The *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

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 1003.1-200x 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.

CX 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.

CX 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*.

RETURN VALUE

Upon successful completion, the *wcstoul*( ) and *wcstoull*( ) functions shall return the converted CX 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].

ERRORS

These functions shall fail if:

CX [EINVAL] The value of *base* is not supported.

[ERANGE] The value to be returned is not representable.

These functions may fail if:

CX [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 1003.1-200x, **<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 *wcstoul*( ) prototype is updated.
- The *wcstoull*( ) function is added.

52984 **NAME**

52985 wcstoumax — convert a wide-character string to an integer type

52986 **SYNOPSIS**

52987 ```
#include <stddef.h>
```

52988 ```
#include <inttypes.h>
```

52989 ```
uintmax_t wcstoumax(const wchar_t *restrict nptr,
```

52990 ```
    wchar_t **restrict endptr, int base);
```

52991 **DESCRIPTION**

52992 Refer to *wcstoimax( )*.

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 1003.1-200x, 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, wcsxfrm_l — wide-character string transformation

SYNOPSIS

```
#include <wchar.h>

size_t wcsxfrm(wchar_t *restrict ws1, const wchar_t *restrict ws2,
    size_t n);
CX  size_t wcsxfrm_l(wchar_t *restrict ws1, const wchar_t *restrict ws2,
    size_t n, locale_t locale);
```

DESCRIPTION

CX For *wcsxfrm*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

CX The *wcsxfrm*( ) and *wcsxfrm_l*( ) functions 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 CX return a value greater than, equal to, or less than 0, corresponding to the result of *wcscoll*( ) and *wcscoll_l*( ) applied to the same two original wide-character strings, and the same *LC_COLLATE* CX category of the locale of the process or the locale object *locale*, respectively. 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.

CX The *wcsxfrm*( ) and *wcsxfrm_l*( ) functions 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 *wcsxfrm*( ) or *wcsxfrm_l*( ), then check *errno*.

RETURN VALUE

CX The *wcsxfrm*( ) and *wcsxfrm_l*( ) functions 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.

CX On error, the *wcsxfrm*( ) and *wcsxfrm_l*( ) functions may set *errno*, but no return value is reserved to indicate an error.

ERRORS

These functions may fail if:

CX [EINVAL] The wide-character string pointed to by *ws2* contains wide-character codes outside the domain of the collating sequence.

The *wcsxfrm_l*( ) function may fail if:

CX [EINVAL] *locale* is not a valid locale object handle.

53061 **EXAMPLES**

53062 None.

53063 **APPLICATION USAGE**

53064 The transformation function is such that two transformed wide-character strings can be ordered
53065 by *wcscmp*( ) as appropriate to collating sequence information in the locale of the process
53066 (category *LC_COLLATE*).

53067 The fact that when *n* is 0 *ws1* is permitted to be a null pointer is useful to determine the size of
53068 the *ws1* array prior to making the transformation.

53069 **RATIONALE**

53070 None.

53071 **FUTURE DIRECTIONS**

53072 None.

53073 **SEE ALSO**

53074 *wcscmp*( ), *wcscoll*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

53075 **CHANGE HISTORY**

53076 First released in Issue 4. Derived from the MSE working draft.

53077 **Issue 5**

53078 Moved from ENHANCED I18N to BASE and the [ENOSYS] error is removed.

53079 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

53080 **Issue 6**

53081 In previous versions, this function was required to return –1 on error.

53082 Extensions beyond the ISO C standard are marked.

53083 The following new requirements on POSIX implementations derive from alignment with the
53084 Single UNIX Specification:

53085 • In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
53086 added if no conversion could be performed.

53087 The *wcsxfrm*( ) prototype is updated for alignment with the ISO/IEC 9899: 1999 standard.

53088 **Issue 7**

53089 The *wcsxfrm_l*( ) function is added from The Open Group Technical Standard, 2006, Extended
53090 API Set Part 4.

53091 **NAME**

53092 wctob — wide-character to single-byte conversion

53093 **SYNOPSIS**

53094 ```
#include <stdio.h>
```

53095 ```
#include <wchar.h>
```

53096 ```
int wctob(wint_t c);
```

53097 **DESCRIPTION**

53098 CX The functionality described on this reference page is aligned with the ISO C standard. Any
53099 conflict between the requirements described here and the ISO C standard is unintentional. This
53100 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

53101 The *wctob*( ) function shall determine whether *c* corresponds to a member of the extended
53102 character set whose character representation is a single byte when in the initial shift state.

53103 The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale.

53104 **RETURN VALUE**

53105 The *wctob*( ) function shall return EOF if *c* does not correspond to a character with length one in
53106 the initial shift state. Otherwise, it shall return the single-byte representation of that character as
53107 an **unsigned char** converted to **int**.

53108 **ERRORS**

53109 No errors are defined.

53110 **EXAMPLES**

53111 None.

53112 **APPLICATION USAGE**

53113 None.

53114 **RATIONALE**

53115 None.

53116 **FUTURE DIRECTIONS**

53117 None.

53118 **SEE ALSO**

53119 *btowc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

53120 **CHANGE HISTORY**

53121 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
53122 (E).
```

NAME

wctomb — convert a wide-character code to a character

SYNOPSIS

```
#include <stdlib.h>

int wctomb(char *s, wchar_t wchar);
```

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 1003.1-200x defers to the ISO C standard.

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.

CX 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 thread-safe. A function that is not required to be thread-safe is not required to be reentrant.

The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-200x 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.

53164 **FUTURE DIRECTIONS**

53165 None.

53166 **SEE ALSO**

53167 *mblen*( ), *mbtowc*( ), *mbstowcs*( ), *wcstombs*( ), the Base Definitions volume of IEEE Std 1003.1-200x,
53168 **<stdlib.h>**

53169 **CHANGE HISTORY**

53170 First released in Issue 4. Derived from the ANSI C standard.

53171 **Issue 6**

53172 Extensions beyond the ISO C standard are marked.

53173 In the DESCRIPTION, a note about reentrancy and thread-safety is added.

NAME

wctrans, wctrans_l — define character mapping

SYNOPSIS

```
#include <wctype.h>

wctrans_t wctrans(const char *charclass);
CX wctrans_t wctrans_l(const char *charclass, locale_t locale);
```

DESCRIPTION

CX For *wctrans*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

CX The *wctrans*( ) and *wctrans_l*( ) functions are 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**.

These functions shall return a value of type **wctrans_t**, which can be used as the second CX argument to subsequent calls of *towctrans*( ) and *towctrans_l*( ).

CX The *wctrans*( ) and *wctrans_l*( ) functions shall determine values of **wctrans_t** according to the rules of the coded character set defined by character mapping information in the locale of the CX process or in the locale represented by *locale*, respectively (category *LC_CTYPE*).

The values returned by *wctrans*( ) shall be valid until a call to *setlocale*( ) that modifies the category *LC_CTYPE*.

CX The values returned by *wctrans_l*( ) shall be valid only in calls to *wctrans_l*( ) with a locale represented by *locale* with the same *LC_CTYPE* category value.

RETURN VALUE

CX The *wctrans*( ) and *wctrans_l*( ) functions 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, they shall return a non-zero object of type **wctrans_t** that can be used in calls to CX *towctrans*( ) and *towctrans_l*( ).

ERRORS

These functions may fail if:

CX [EINVAL] The character mapping name pointed to by *charclass* is not valid in the current locale.

The *wctrans_l*( ) function may fail if:

CX [EINVAL] *locale* is not a valid locale object handle.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

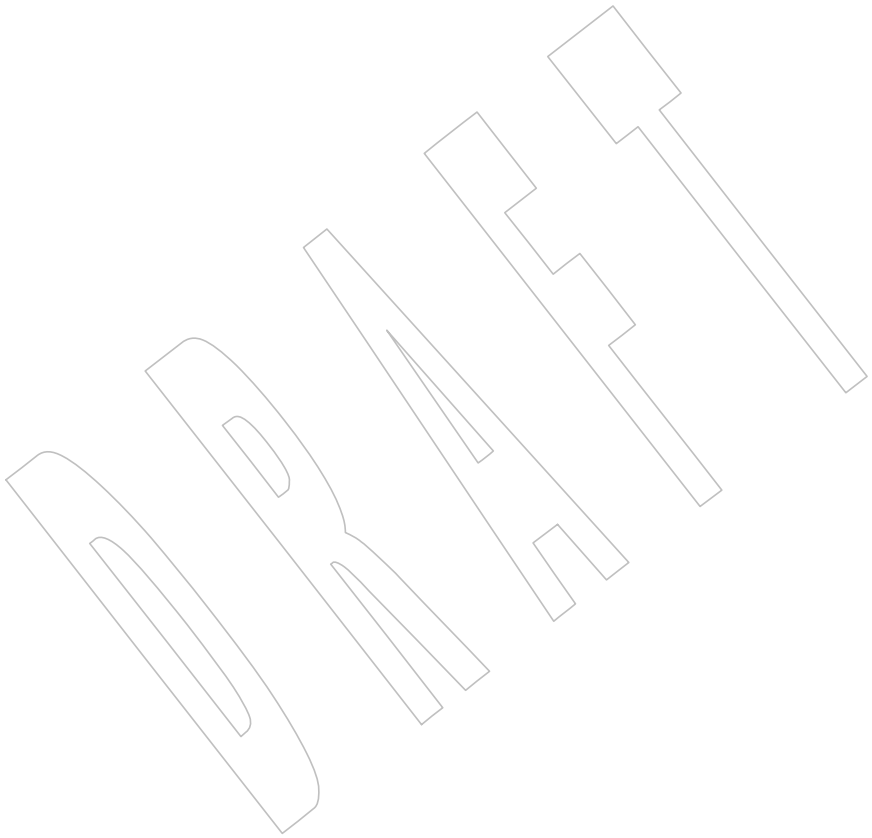
towctrans( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wctype.h>**

CHANGE HISTORY

First released in Issue 5. Derived from ISO/IEC 9899: 1990/Amendment 1: 1995 (E).

Issue 7

The *wctrans_l( )* function is added from The Open Group Technical Standard, 2006, Extended API Set Part 4.



NAME

wctype, wctype_l — define character class

SYNOPSIS

```
#include <wctype.h>

wctype_t wctype(const char *property);
CX wctype_t wctype_l(const char *property, locale_t locale);
```

DESCRIPTION

CX For *wctype*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

CX The *wctype*( ) and *wctype_l*( ) functions are defined for valid character class names as defined in CX the current locale or in the locale represented by *locale*, respectively.

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.

These functions shall return a value of type **wctype_t**, which can be used as the second CX argument to subsequent calls of *iswctype*( ) and *iswctype_l*( ).

CX The *wctype*( ) and *wctype_l*( ) functions shall determine values of **wctype_t** according to the rules of the coded character set defined by character type information in the locale of the process CX or in the locale represented by *locale*, respectively (category *LC_CTYPE*).

The values returned by *wctype*( ) shall be valid until a call to *setlocale*( ) that modifies the category *LC_CTYPE*.

CX The values returned by *wctype_l*( ) shall be valid only in calls to *iswctype_l*( ) with a locale represented by *locale* with the same *LC_CTYPE* category value.

RETURN VALUE

CX The *wctype*( ) and *wctype_l*( ) functions shall return 0 if the given character class name is not valid for the current locale (category *LC_CTYPE*); otherwise, they shall return an object of type CX **wctype_t** that can be used in calls to *iswctype*( ) and *iswctype_l*( ).

ERRORS

The *wctype_l*( ) function may fail if:

CX [EINVAL] *locale* is not a valid locale object handle.

53259 **EXAMPLES**

53260 None.

53261 **APPLICATION USAGE**

53262 None.

53263 **RATIONALE**

53264 None.

53265 **FUTURE DIRECTIONS**

53266 None.

53267 **SEE ALSO**

53268 *iswctype( )*, the Base Definitions volume of IEEE Std 1003.1-200x, **<wctype.h>**

53269 **CHANGE HISTORY**

53270 First released in Issue 4.

53271 **Issue 5**

53272 The following change has been made in this issue for alignment with
53273 ISO/IEC 9899: 1990/Amendment 1: 1995 (E):

53274 • The SYNOPSIS has been changed to indicate that this function and associated data types
53275 are now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

53276 **Issue 7**

53277 The *wctype_l( )* function is added from The Open Group Technical Standard, 2006, Extended API
53278 Set Part 4.

53279 **NAME**

53280 wcwidth — number of column positions of a wide-character code

53281 **SYNOPSIS**

53282 XSI
```
#include <wchar.h>

int wcwidth(wchar_t wc);
```
53283

53284 **DESCRIPTION**

53285 The *wcwidth*( ) function shall determine the number of column positions required for the wide
53286 character *wc*. The application shall ensure that the value of *wc* is a character representable as a
53287 **wchar_t**, and is a wide-character code corresponding to a valid character in the current locale.

53288 **RETURN VALUE**

53289 The *wcwidth*( ) function shall either return 0 (if *wc* is a null wide-character code), or return the
53290 number of column positions to be occupied by the wide-character code *wc*, or return –1 (if *wc*
53291 does not correspond to a printable wide-character code).

53292 **ERRORS**

53293 No errors are defined.

53294 **EXAMPLES**

53295 None.

53296 **APPLICATION USAGE**

53297 This function was removed from the final ISO/IEC 9899:1990/Amendment 1:1995 (E), and the
53298 return value for a non-printable wide character is not specified.

53299 **RATIONALE**

53300 None.

53301 **FUTURE DIRECTIONS**

53302 None.

53303 **SEE ALSO**

53304 *wcswidth*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

53305 **CHANGE HISTORY**

53306 First released as a World-wide Portability Interface in Issue 4. Derived from the MSE working
53307 draft.

53308 **Issue 6**

53309 The Open Group Corrigendum U021/12 is applied. This function is marked as an extension.

53310 The normative text is updated to avoid use of the term "must" for application requirements.

NAME

wmemchr — find a wide character in memory

SYNOPSIS

```
#include <wchar.h>

wchar_t *wmemchr(const wchar_t *ws, wchar_t wc, 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 1003.1-200x 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( ), *wmemcpy*( ), *wmemmove*( ), *wmemset*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

The normative text is updated to avoid use of the term "must" for application requirements.

53347 **NAME**

53348 wmemcmp — compare wide characters in memory

53349 **SYNOPSIS**

53350
```
#include <wchar.h>
```

53351
```
int wmemcmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);
```

53352 **DESCRIPTION**

53353 CX The functionality described on this reference page is aligned with the ISO C standard. Any
53354 conflict between the requirements described here and the ISO C standard is unintentional. This
53355 volume of IEEE Std 1003.1-200x defers to the ISO C standard.

53356 The *wmemcmp*( ) function shall compare the first *n* wide characters of the object pointed to by
53357 *ws1* to the first *n* wide characters of the object pointed to by *ws2*. This function shall not be
53358 affected by locale and all **wchar_t** values shall be treated identically. The null wide character and
53359 **wchar_t** values not corresponding to valid characters shall not be treated specially.

53360 If *n* is zero, the application shall ensure that *ws1* and *ws2* are valid pointers, and the function
53361 shall behave as if the two objects compare equal.

53362 **RETURN VALUE**

53363 The *wmemcmp*( ) function shall return an integer greater than, equal to, or less than zero,
53364 respectively, as the object pointed to by *ws1* is greater than, equal to, or less than the object
53365 pointed to by *ws2*.

53366 **ERRORS**

53367 No errors are defined.

53368 **EXAMPLES**

53369 None.

53370 **APPLICATION USAGE**

53371 None.

53372 **RATIONALE**

53373 None.

53374 **FUTURE DIRECTIONS**

53375 None.

53376 **SEE ALSO**

53377 *wmemchr*( ), *wmemcpy*( ), *wmemmove*( ), *wmemset*( ), the Base Definitions volume of
53378 IEEE Std 1003.1-200x, **<wchar.h>**

53379 **CHANGE HISTORY**

53380 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
53381 (E).

53382 **Issue 6**

53383 The normative text 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

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 1003.1-200x 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 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

The normative text 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

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 1003.1-200x 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*( ), t,

NAME

wmemset — set wide characters in memory

SYNOPSIS

```
#include <wchar.h>

wchar_t *wmemset(wchar_t *ws, wchar_t wc, 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 1003.1-200x 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 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

The normative text 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 1003.1-200x, Section 2.6, Word Expansions, subject to quoting as in the Shell and Utilities volume of IEEE Std 1003.1-200x, 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 1003.1-200x, Section 2.6.3, Command Substitution. It also shall not contain unquoted parentheses or braces, except in the context of command or variable substitution. The application shall ensure that every member of *words* which it expects to have expanded by *wordexp*( ) does not contain an unquoted initial comment character. The application shall also ensure that any words which it intends to be ignored (because they begin or continue a comment) are deleted from *words*. If the argument *words* contains an unquoted comment character (number sign) that is the beginning of a token, *wordexp*( ) shall either treat the comment character as a regular character, or interpret it as a comment indicator and ignore the remainder of *words*.

The structure type **wordexp_t** is defined in the **<wordexp.h>** header and includes at least the following members:

Member Type	Member Name	Description
size_t	*we_wordc*	Count of words matched by *words*.
**char ** **	*we_wordv*	Pointer to list of expanded words.
size_t	*we_offs*	Slots to reserve at the beginning of *pwordexp*−>*we_wordv*.

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 1003.1-200x, Section 2.6.5, Field Splitting) or pathname expansion (see the Shell and Utilities volume of IEEE Std 1003.1-200x, 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 1003.1-200x, 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 1003.1-200x, Section 2.5.2, Special Parameters is unspecified.

It is the caller's responsibility to allocate the storage pointed to by *pwordexp*. The *wordexp*( ) function shall allocate other space as needed, including memory pointed to by *pwordexp*−>*we_wordv*. The *wordfree*( ) function frees any memory associated with *pwordexp* from a previous call to *wordexp*( ).

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>**:

53542 WRDE_APPEND Append words generated to the ones from a previous call to *wordexp*( ).

53543 WRDE_DOOFFS Make use of *pwordexp–>we_offs*. If this flag is set, *pwordexp–>we_offs* is
53544 used to specify how many null pointers to add to the beginning of
53545 *pwordexp–>we_wordv*. In other words, *pwordexp–>we_wordv* shall point to
53546 *pwordexp–>we_offs* null pointers, followed by *pwordexp–>we_wordc* word
53547 pointers, followed by a null pointer.

53548 WRDE_NOCMD If the implementation supports the utilities defined in the Shell and
53549 Utilities volume of IEEE Std 1003.1-200x, fail if command substitution, as
53550 specified in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section
53551 2.6.3, Command Substitution, is requested.

53552 WRDE_REUSE The *pwordexp* argument was passed to a previous successful call to
53553 *wordexp*( ), and has not been passed to *wordfree*( ). The result shall be the
53554 same as if the application had called *wordfree*( ) and then called *wordexp*( )
53555 without WRDE_REUSE.

53556 WRDE_SHOWERR Do not redirect *stderr* to **/dev/null**.

53557 WRDE_UNDEF Report error on an attempt to expand an undefined shell variable.

53558 The WRDE_APPEND flag can be used to append a new set of words to those generated by a
53559 previous call to *wordexp*( ). The following rules apply to applications when two or more calls to
53560 *wordexp*( ) are made with the same value of *pwordexp* and without intervening calls to *wordfree*( ):

53561 1. The first such call shall not set WRDE_APPEND. All subsequent calls shall set it.

53562 2. All of the calls shall set WRDE_DOOFFS, or all shall not set it.

53563 3. After the second and each subsequent call, *pwordexp–>we_wordv* shall point to a list
53564 containing the following:

53565 a. Zero or more null pointers, as specified by WRDE_DOOFFS and
53566 *pwordexp–>we_offs*

53567 b. Pointers to the words that were in the *pwordexp–>we_wordv* list before the call, in
53568 the same order as before

53569 c. Pointers to the new words generated by the latest call, in the specified order

53570 4. The count returned in *pwordexp–>we_wordc* shall be the total number of words from all of
53571 the calls.

53572 5. The application can change any of the fields after a call to *wordexp*( ), but if it does it shall
53573 reset them to the original value before a subsequent call, using the same *pwordexp* value,
53574 to *wordfree*( ) or *wordexp*( ) with the WRDE_APPEND or WRDE_REUSE flag.

53575 If the implementation supports the utilities defined in the Shell and Utilities volume of
53576 IEEE Std 1003.1-200x, and *words* contains an unquoted character—<newline>, `'|'`, `'&'`, `';'`,
53577 `'<'`, `'>'`, `'('`, `')'`, `'{'`, `'}'`—in an inappropriate context, *wordexp*( ) shall fail, and the number
53578 of expanded words shall be 0.

53579 Unless WRDE_SHOWERR is set in *flags*, *wordexp*( ) shall redirect *stderr* to **/dev/null** for any
53580 utilities executed as a result of command substitution while expanding *words*. If
53581 WRDE_SHOWERR is set, *wordexp*( ) may write messages to *stderr* if syntax errors are detected
53582 while expanding *words*.

53583 The application shall ensure that if WRDE_DOOFFS is set, then *pwordexp–>we_offs* has the same
53584 value for each *wordexp*( ) call and *wordfree*( ) call using a given *pwordexp*.

53585 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:

```
(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

53632 substitution. In any case, it would have to scan *words* for unquoted special characters.

53633 Another proposal is to add the following options to *sh*:

53634 **–w** *wordlist*
53635 This option provides a wordlist expansion service to applications. The words in *wordlist*
53636 shall be expanded and the following written to standard output:

53637 1. The count of the number of words after expansion, in decimal, followed by a null
53638 byte

53639 2. The number of bytes needed to represent the expanded words (not including null
53640 separators), in decimal, followed by a null byte

53641 3. The expanded words, each terminated by a null byte

53642 If an error is encountered during word expansion, *sh* exits with a non-zero status after
53643 writing the former to report any words successfully expanded

53644 **–P** Run in "protected" mode. If specified with the **–w** option, no command substitution shall
53645 be performed.

53646 With these options, *wordexp*() could be implemented fairly simply by creating a subprocess
53647 using *fork*() and executing *sh* using the line:

53648 `execl(<shell path>, "sh", "-P", "-w", words, (char *)0);`

53649 after directing standard error to **/dev/null**.

53650 It seemed objectionable for a library routine to write messages to standard error, unless explicitly
53651 requested, so *wordexp*() is required to redirect standard error to **/dev/null** to ensure that no
53652 messages are generated, even for commands executed for command substitution. The
53653 WRDE_SHOWERR flag can be specified to request that error messages be written.

53654 The WRDE_REUSE flag allows the implementation to avoid the expense of freeing and
53655 reallocating memory, if that is possible. A minimal implementation can call *wordfree*() when
53656 WRDE_REUSE is set.

53657 **FUTURE DIRECTIONS**
53658 None.

53659 **SEE ALSO**
53660 *fnmatch*(), *glob*(), the Base Definitions volume of IEEE Std 1003.1-200x, **<wordexp.h>**, the Shell
53661 and Utilities volume of IEEE Std 1003.1-200x, Chapter 2, Shell Command Language

53662 **CHANGE HISTORY**
53663 First released in Issue 4. Derived from the ISO POSIX-2 standard.

53664 **Issue 5**
53665 Moved from POSIX2 C-language Binding to BASE.

53666 **Issue 6**
53667 The normative text is updated to avoid use of the term "must" for application requirements.

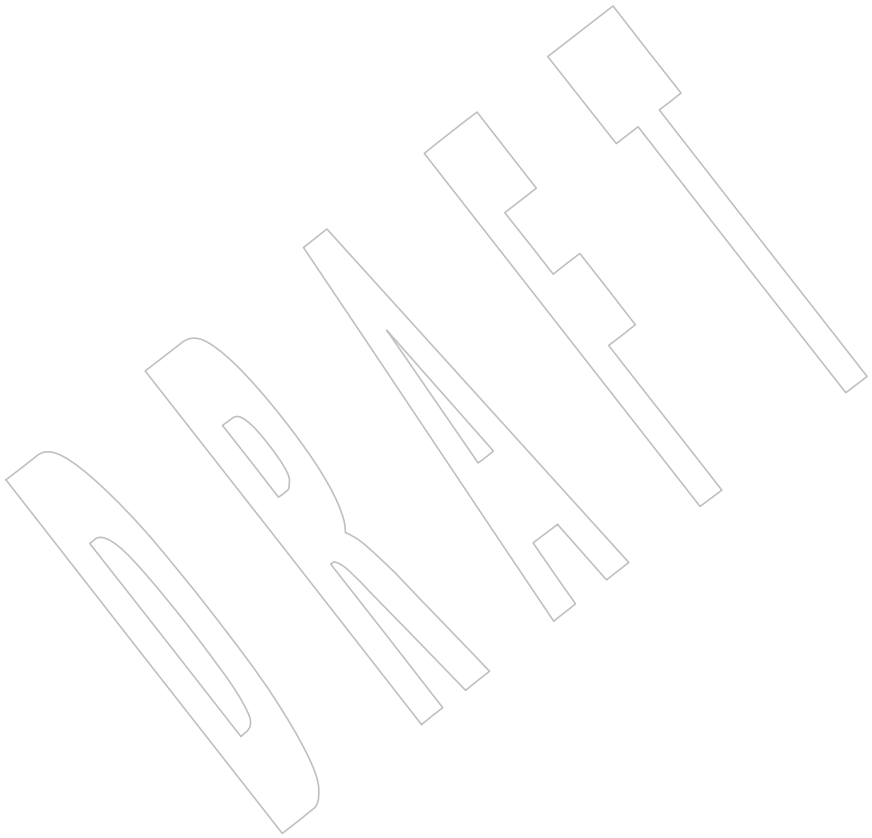
53668 The **restrict** keyword is added to the *wordexp*() prototype for alignment with the
53669 ISO/IEC 9899: 1999 standard.

53670 **NAME**

53671 wprintf — print formatted wide-character output

53672 **SYNOPSIS**

53673 ```
#include <stdio.h>
```

53674 ```
#include <wchar.h>
```

53675 ```
int wprintf(const wchar_t *restrict format, ...);
```

53676 **DESCRIPTION**

53677 Refer to *fwprintf( )*.



  
```

NAME

pwrite, write — write on a file

SYNOPSIS

```
#include <unistd.h>

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 the position of the last byte written is greater than or equal to the length of the file, the length of the file shall be set to this position plus one.

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 file size limit of the process 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, *write*( ) requests shall be handled differently, in the following ways:
  - The *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, *write*( ) shall transfer all the data and return the number of bytes requested. Otherwise, *write*( ) shall transfer no data and return –1 with *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 *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, *write*( ) shall block the calling thread until the data can be accepted.
- If the O_NONBLOCK flag is set, *write*( ) shall not block the thread. If some data can be written without blocking the thread, *write*( ) shall write what it can and return the number of bytes written. Otherwise, it shall return –1 and set *errno* to [EAGAIN].

Upon successful completion, where *nbyte* is greater than 0, *write*( ) shall mark for update the *st_ctime* and *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 *fildes*.

If *fildes* refers to a socket, *write*( ) shall be equivalent to *send*( ) with no flags set.

SIO 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.

SHM If *fildes* refers to a shared memory object, the result of the *write*( ) function is unspecified.

TYM If *fildes* refers to a typed memory object, the result of the *write*( ) function is unspecified.

OB XSR If *fildes* refers to a STREAM, the operation of *write*( ) shall be determined by the values of the minimum and maximum *nbyte* range (packet size) accepted by the STREAM. These values are determined by the topmost STREAM module. If *nbyte* falls within the packet size range, *nbyte*

bytes shall be written. If *nbyte* does not fall within the range and the minimum packet size value is 0, *write*( ) shall break the buffer into maximum packet size segments prior to sending the data downstream (the last segment may contain less than the maximum packet size). If *nbyte* does not fall within the range and the minimum value is non-zero, *write*( ) shall fail with *errno* set to [ERANGE]. Writing a zero-length buffer (*nbyte* is 0) to a STREAMS device sends 0 bytes with 0 returned. However, writing a zero-length buffer to a STREAMS-based pipe or FIFO sends no message and 0 is returned. The process may issue I_SWROPT *ioctl*( ) to enable zero-length messages to be sent across the pipe or FIFO.

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 and does not change the file offset (regardless of whether O_APPEND is set). 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, these functions 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

These 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] XSI An attempt was made to write a file that exceeds the implementation-defined maximum file size or the file size limit of the process, 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*.

[EINTR] 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 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.

OB XSR [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, a SIGPIPE signal shall also be sent to the thread.

These functions may fail if:

OB XSR [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.

[ENOBUFS] 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.

OB XSR [ENXIO] A hangup occurred on the STREAM being written to.

OB XSR 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:

XSI [EINVAL] The *offset* argument is invalid. The value is negative.

XSI [ESPIPE] *fildes* is associated with a pipe or FIFO.

EXAMPLES

53847 **Writing from a Buffer**

53848 The following example writes data from the buffer pointed to by *buf* to the file associated with
53849 the file descriptor *fd*.

```
53850 #include <sys/types.h>
53851 #include <string.h>
53852 ...
53853 char buf[20];
53854 size_t nbytes;
53855 ssize_t bytes_written;
53856 int fd;
53857 ...
53858 strcpy(buf, "This is a test\n");
53859 nbytes = strlen(buf);
53860 bytes_written = write(fd, buf, nbytes);
53861 ...
```

53862 **APPLICATION USAGE**

53863 None.

53864 **RATIONALE**

53865 See also the RATIONALE section in *read( )*.

53866 An attempt to write to a pipe or FIFO has several major characteristics:

53867 • *Atomic/non-atomic*: A write is atomic if the whole amount written in one operation is not
53868 interleaved with data from any other process. This is useful when there are multiple
53869 writers sending data to a single reader. Applications need to know how large a write
53870 request can be expected to be performed atomically. This maximum is called {PIPE_BUF}.
53871 This volume of IEEE Std 1003.1-200x does not say whether write requests for more than
53872 {PIPE_BUF} bytes are atomic, but requires that writes of {PIPE_BUF} or fewer bytes shall
53873 be atomic.

53874 • *Blocking/immediate*: Blocking is only possible with O_NONBLOCK clear. If there is enough
53875 space for all the data requested to be written immediately, the implementation should do
53876 so. Otherwise, the calling thread may block; that is, pause until enough space is available
53877 for writing. The effective size of a pipe or FIFO (the maximum amount that can be written
53878 in one operation without blocking) may vary dynamically, depending on the
53879 implementation, so it is not possible to specify a fixed value for it.

53880 • *Complete/partial/deferred*: A write request:

```
53881 int fildes;
53882 size_t nbyte;
53883 ssize_t ret;
53884 char *buf;
53885 ret = write(fildes, buf, nbyte);
```

53886 may return:

53887 Complete *ret=nbyte*

53888 Partial *ret<nbyte*

53889 This shall never happen if *nbyte*≤{PIPE_BUF}. If it does happen (with
53890 *nbyte*>{PIPE_BUF}), this volume of IEEE Std 1003.1-200x does not guarantee
53891 atomicity, even if *ret*≤{PIPE_BUF}, because atomicity is guaranteed according
53892 to the amount *requested*, not the amount *written*.

Deferred: *ret*=–1, *errno*=[EAGAIN]

This error indicates that a later request may succeed. It does not indicate that it *shall* succeed, even if *nbyte*≤{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 [EAGAIN] is caused by a particular value of *nbyte*>{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*			
Immediately Writable:	**None**	**Some**	*nbyte*
nbyte≤{PIPE_BUF}	Atomic blocking *nbyte*	Atomic blocking *nbyte*	Atomic immediate *nbyte*
nbyte>{PIPE_BUF}	Blocking *nbyte*	Blocking *nbyte*	Blocking *nbyte*

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 *fcntl*( )), a write request shall never block.

Write to a Pipe or FIFO with O_NONBLOCK *set*			
Immediately Writable:	**None**	**Some**	*nbyte*
nbyte≤{PIPE_BUF}	–1, [EAGAIN]	–1, [EAGAIN]	Atomic *nbyte*
nbyte>{PIPE_BUF}	–1, [EAGAIN]	<*nbyte* or –1, [EAGAIN]	≤*nbyte* or –1, [EAGAIN]

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 1003.1-200x 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 {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, {PIPE_BUF} bytes must be written; if not, at least some progress must have been made.

Where this volume of IEEE Std 1003.1-200x requires –1 to be returned and *errno* set to [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 1003.1-200x). The error indications in this volume of IEEE Std 1003.1-200x were chosen so that an application can distinguish these cases from end-of-file. While *write*( ) cannot receive an indication of end-of-file, *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 *write*( ) indicates a successful write of an end-of-file indication.

Implementations are allowed, but not required, to perform error checking for *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 *read*( ).

Writes can be serialized with respect to other reads and writes. If a *read*( ) of file data can be proven (by any means) to occur after a *write*( ) of the data, it must reflect that *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 *write*( ) calls to subsequent *read*( ) calls. This requirement is particularly significant for networked file systems, where some caching schemes violate these semantics.

Note that this is specified in terms of *read*( ) and *write*( ). The XSI extensions *readv*( ) and *writev*( ) also obey these semantics. A new "high-performance" write analog that did not follow these serialization requirements would also be permitted by this wording. This volume of IEEE Std 1003.1-200x is also silent about any effects of application-level caching (such as that done by *stdio*).

This volume of IEEE Std 1003.1-200x 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.

This volume of IEEE Std 1003.1-200x does not specify behavior of concurrent writes to a file from multiple processes. Applications should use some form of concurrency control.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod( ), *creat*( ), *dup*( ), *fcntl*( ), *getrlimit*( ), *lseek*( ), *open*( ), *pipe*( ), *ulimit*( ), *writev*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<limits.h>**, **<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 *pwrite*( ) function is added.

Issue 6

The DESCRIPTION states that the *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 made optional.

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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/146 is applied, updating text in the ERRORS section from "a SIGPIPE signal is generated to the calling process" to "a SIGPIPE signal shall also be sent to the thread".

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/147 is applied, making a correction to the RATIONALE.

Issue 7

The *pwrite*( ) function is moved from the XSI option to the Base.

Functionality relating to the XSI STREAMS option is marked obsolescent.

SD5-XSH-ERN-160 is applied, updating the DESCRIPTION to clarify the requirements for the *pwrite*( ) function, and to change the use of the phrase "file pointer" to "file offset".

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];
```

```
        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 1003.1-200x, **<limits.h>**, **<sys/uio.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 6

Split out from the *write*( ) reference page.

54073 **NAME**

54074 wscanf — convert formatted wide-character input

54075 **SYNOPSIS**

54076 ```
#include <stdio.h>
```

54077 ```
#include <wchar.h>
```

54078 ```
int wscanf(const wchar_t *restrict format, ...);
```

54079 **DESCRIPTION**

54080 Refer to *fwscanf( )*.
```

NAME

y0, y1, yn — Bessel functions of the second kind

SYNOPSIS

XSI

```
#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 pole 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.

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 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

feclearexcept( ), *fetestexcept( )*, *isnan( )*, *j0( )*, the Base Definitions volume of IEEE Std 1003.1-200x, 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 normative text 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.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/148 is applied, updating the RETURN VALUE and ERRORS sections. The changes are made for consistency with the general rules stated in "Treatment of Error Conditions for Mathematical Functions" in the Base Definitions volume of IEEE Std 1003.1-200x.

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index