International Standard ISO/IEC 9945-1: 1996 (E)
IEEE Std 1003.1, 1996 Edition
(Incorporating ANSI/IEEE Stds 1003.1-1990, 1003.1b-1993,
1003.1c-1995, and 1003.1i-1995)

Information technology—Portable Operating System Interface (POSIX®)—Part 1: System Application Program Interface (API) [C Language]

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

Adopted as an International Standard by the International Organization for Standardization and by the International Electrotechnical Commission

Published by The Institute of Electrical and Electronics Engineers, Inc.

Abstract: This standard is part of the POSIX series of standards for applications and user interfaces to open systems. It defines the applications interface to basic system services for input/output, file system access, and process management. It also defines a format for data interchange. When options specified in the Realtime Extension are included, the standard also defines interfaces appropriate for realtime applications. When options specified in the Threads Extension are included, the standard also defines interfaces appropriate for multithreaded applications. This standard is stated in terms of its C language binding.
Keywords: API, application portability, C (programming language), data processing, information interchange, open systems, operating system, portable application, POSIX, programming language, realtime, system configuration computer interface, threads

The Institute of Electrical and Electronics Engineers, Inc. 345 East 47th Street, New York, NY 10017-2394, USA

ISBN 1-55937-573-6

12 July 1996

SH94352

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and nongovernmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75% of the national bodies casting a vote.

International Standard ISO/IEC 9945-1 :1996 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*. This edition cancels and replaces the first edition (ISO/IEC 9945-1 :1990), which has been technically revised. This edition also consolidates Amendments 1 and 2 and Technical Corrigendum 1 to ISO/IEC 9945-1 :1990.

ISO/IEC 9945 consists of the following parts, under the general title *Information technology—Portable operating system interface (POSIX)*:

— *Part 1.: System application program interface (API) [C language]*
— *Part 2.: Shell and utilities*
— *Part 3.: System administration* (under development)

Annexes A to H of ISO/IEC 9945-1 are provided for information only.

Introduction

(This introduction is not a normative part of ISO/IEC 9945-1, Information technology—Portable Operating System Interface (POSIX)—Part 1.: System Application Programming Interface (API) [C Language])

The purpose of this part of ISO/IEC 9945 is to define a standard operating system interface and environment based on the UNIX[1] Operating System documentation to support application portability at the source level. This is intended for systems implementors and applications software developers.

Initially,[2] the focus of this part of ISO/IEC 9945 is to provide standardized services via a C language interface. Future revisions are expected to contain bindings for other programming languages as well as for the C language. This will be accomplished by breaking this part of ISO/IEC 9945 into multiple portions—one defining core requirements independent of any programming language, and others composed of programming language bindings.

The core requirements portion will define a set of required services common to any programming language that can be reasonably expected to form a language binding to this part of ISO/IEC 9945. These services will be described in terms of functional requirements and will not define programming language-dependent interfaces. Language bindings will consist of two major parts. One will contain the programming language's standardized interface for accessing the core services defined in the programming language-independent core requirements section of this part of ISO/IEC 9945. The other will contain a standardized interface for language-specific services. Any implementation claiming conformance to this part of ISO/IEC 9945 with any language binding will be required to comply with both sections of the language binding.

Within this document, the term “POSIX.1” refers to this part of ISO/IEC 9945 itself.

Organization of This Part of ISO/IEC 9945

This part of ISO/IEC 9945 is divided into four elements:

1) Statement of scope and list of normative references (Section 1.)
2) Definitions and global concepts (Section 2.)
3) The various interface facilities (Sections 3. through 9. and 11. through 18.)
4) Data interchange format (Section 10.)

Most of the sections describe a single service interface. The C Language binding for the service interface is given in the subclause labeled Synopsis. The Description subclause provides a specification of the operation performed by the service interface. Some examples may be provided to illustrate the interfaces described. In most cases there are also Returns and Errors subclauses specifying return values and possible error conditions. References are used to direct the reader to other related sections. Additional material to complement sections in this part of ISO/IEC 9945 may be found in the Rationale and Notes, Annex B. This annex provides historical perspectives into the technical choices made by the developers of this part of ISO/IEC 9945. It also provides information to emphasize consequences of the interfaces described in the corresponding section of this part of ISO/IEC 9945.

Informative annexes are not part of the standard and are provided for information only. (There is a type of annex called “normative” that is part of a standard and imposes requirements, but there are currently no such normative annexes in this part of ISO/IEC 9945.) They are provided for guidance and to help understanding.

In publishing this part of ISO/IEC 9945, its developers simply intend to provide a yardstick against which various operating system implementations can be measured for conformance. It is *not* the intent of the developers to measure or rate any products, to reward or sanction any vendors of products for conformance or lack of conformance to this part

[1]UNIX is a registered trademark in the USA and other countries licensed exclusively through X/Open.
[2]The vertical rules in the right margin depict technical or significant non-editorial changes from ISO/IEC 9945 :1-1990. A vertical rule beside an empty line indicates deleted text.

of ISO/IEC 9945, or to attempt to enforce this part of ISO/IEC 9945 by these or any other means. The responsibility for determining the degree of conformance or lack thereof with this part of ISO/IEC 9945 rests solely with the individual who is evaluating the product claiming to be in conformance with this part of ISO/IEC 9945 .

Base Documents

The various interface facilities described herein are based on the *1984 /usr/group Standard* derived and published by the UniForum (formerly/usr/group) Standards Committee. The *1984 /usr/group Standard* and this part of ISO/IEC 9945 are largely based on UNIX Seventh Edition, UNIX System III, UNIX System V, 4.2BSD, and 4.3BSD documentation,[3] but wherever possible, compatibility with other systems derived from the UNIX operating system, or systems compatible with that system, has been maintained.

Background

The developers of POSIX.1 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. In the course of their deliberations, the developers reviewed related American and international standards, both published and in progress.

Conceptually, POSIX.1 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, that establishes standard semantics and syntax. Since this interface enables application writers to write portable applications—it was developed with that goal in mind—it has been designated POSIX,[4] an acronym for Portable Operating System Interface.

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

Audience

The intended audience for ISO/IEC 9945 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 part of ISO/IEC 9945:

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

[3]The standard developers are grateful to both AT&T and UniForum for permission to use their materials.
[4]The name POSIX was suggested by Richard Stallman. It is expected to be pronounced *pahz-icks*, as in *positive*, not *poh-six*, or other variations. The pronunciation has been published in an attempt to promulgate a standardized way of referring to a standard operating system interface.

operating system based on the UNIX system documentation. This part of ISO/IEC 9945 codifies the common, existing definition of the UNIX system. There was no attempt to define a new system interface.

Interface, Not Implementation

This part of ISO/IEC 9945 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 Annex B). Symbolic names are given for constants (such as signals and error numbers) rather than numbers.

Source, Not Object, Portability

This part of ISO/IEC 9945 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 part of ISO/IEC 9945 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. However, few impediments were placed in the way of binary compatibility, and some remarks on this are found in Annex B. See B.1.3.1.1 and B.4.8.

The C Language

This part of ISO/IEC 9945 is written in terms of the standard C language as specified in the C Standard {2}.[5] See B.1.3 and B.1.1.1.

No Super-User, No System Administration

There was no intention to specify all aspects of an operating system. System administration facilities and functions are excluded from POSIX.1, and functions usable only by the super-user have not been included. Annex B notes several such instances. Still, an implementation of the standard interface may also implement features not in this part of ISO/IEC 9945; see 1.3.1.1. This part of ISO/IEC 9945 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, POSIX.1 is as minimal as possible. For example, it usually specifies only one set of functions to implement a capability. Exceptions were made in some cases where long tradition and many existing applications included certain functions, such as *creat*(). In such cases, as throughout POSIX.1, redundant definitions were avoided: *creat*() is defined as a special case of *open*(). Redundant functions or implementations with less tradition were excluded.

Broadly Implementable

The developers of POSIX.1 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

[5]The number in braces corresponds to those of the references in 1.2 (or the bibliographic entry in Annex A if the number is preceded by the letter B).

No direct references to this goal appear in this part of ISO/IEC 9945 , but some results of it are mentioned in Annex B.

Minimal Changes to Historical Implementations

There are no known historical implementations that will not have to change in some area to conform to this part of ISO/IEC 9945, and in a few areas POSIX.1 does not exactly match any existing system interface (for example, see the discussion of O_NONBLOCK in B.6). Nonetheless, there is a set of functions, types, definitions, and concepts that form an interface that is common to most historical implementations. POSIX.1 specifies that common interface and extends it in areas where there has historically been no consensus, preferably

1) By standardizing an interface like one in an historical implementation; e.g., directories, or;
2) By specifying an interface that is readily implementable in terms of, and backwards compatible with, historical implementations, such as the extended tar format in 10.1.1, or;
3) By specifying an interface that, when added to an historical implementation, will not conflict with it, like B.6.

Required changes to historical implementations have been kept to a minimum, but they do exist, and Annex B points out some of them.

POSIX.1 is specifically not a codification of a particular vendor's product. It is similar to the UNIX system, but it is not identical to it.

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 POSIX.1. These need to be used with careful management to be able to adapt to future POSIX.1 extensions and/or port to implementations that provide these services in a different manner.

Minimal Changes to Existing Application Code

A goal of POSIX.1 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. Annex B points out the major places where POSIX.1 implies such changes.

Realtime Extension

This portion of this part of ISO/IEC 9945 defines optional sets of systems interfaces to support the source portability of applications with realtime requirements. The interfaces included in this standard were the set required to make POSIX.1 minimally usable to realtime applications on single processor systems. The scope is to take existing realtime operating system practice and add it to the base standard.

The definition of *realtime* used in defining the scope of this standard is:

> “Realtime in operating systems: the ability of the operating system to provide a required level of service in a bounded response time.”

The key elements of defining the scope are

1) Defining a sufficient set of functionality to cover a significant part of the realtime application program domain, and
2) Defining sufficient performance constraints and performance-related functions to allow a realtime application to achieve deterministic response from the system.

Specifically within the scope is to define interfaces that do not preclude high-performance implementations on traditional uniprocessor realtime systems.

Wherever possible, the requirements of other application environments were included in this interface definition. The specific areas are noted in the scope overviews of each of the interface areas given below. It is beyond the scope of these interfaces to support networking or multiprocessor functionality.

The specific functional areas included in this standard and their scope includes:

- *Semaphores*: A minimum synchronization primitive to serve as a basis for more complex synchronization mechanisms to be defined by the application program.
- *Process memory locking*: A performance improvement facility to bind application programs into the high-performance random access memory of a computer system. This avoids potential latencies introduced by the operating system in storing parts of a program that were not recently referenced on secondary memory devices.
- *Memory mapped files and shared memory*: A performance improvement facility to allow for programs to access files as part of the program images and for separate application programs to have portions of their program image commonly accessible.
- *Priority scheduling*: A performance and determinism improvement facility to allow applications to determine the order in which processes that are ready to run are granted access to processor resources.
- *Realtime signal extension*: A determinism improvement facility that augments the signals mechanism of historical implementations to enable asynchronous signal notifications to an application to be queued without impacting compatibility with the existing signals interface.
- *Timers*: A functionality and determinism improvement facility to increase the resolution and capabilities of the time-base interface.
- *Interprocess communication*: A functionality enhancement to add a high-performance, deterministic interprocess communication facility for local communication. Network transparency is beyond the scope of this interface.
- *Synchronized input and output*: A determinism and robustness improvement mechanism to enhance the data input and output mechanisms so that an application can insure that the data being manipulated is physically present on secondary mass storage devices.
- *Asynchronous input and output*: A functionality enhancement to allow an application process to queue data input and output commands with asynchronous notification of completion. This facility includes in its scope the requirements of supercomputer applications.

Threads Extension

This portion of this part of ISO/IEC 9945 provides the base standard with interfaces and functionality to support multiple flows of control, called *threads*, within a process. The facilities provided represent a small set of syntactic and semantic extensions to POSIX.1 in order to support a convenient interface for multithreading functions.

The key elements defining the scope are

- Defining a sufficient set of functionality to support multiple threads of control within a process
- Defining a sufficient set of functionality to support the realtime application domain
- Defining sufficient performance constraints and performance-related functions to allow a realtime application to achieve deterministic response from the system

Wherever possible, the requirements of other application environments were included in the interface definition. The interfaces in this standard are specifically targeted at supporting tightly coupled multitasking environments, including multiprocessors and advanced language constructs.

The specific functional areas covered by this standard and their scopes 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.
- *Harmonization*: with the existing POSIX.1 interfaces.

Related Standards Activities

Activities to extend this part of ISO/IEC 9945 to address additional requirements are in progress, and similar efforts can be anticipated in the future.

The following areas are under active consideration at this time, or are expected to become active in the near future:[6]

1) Language-independent service descriptions of this part of ISO/IEC 9945
2) C, Ada, and FORTRAN language bindings to (1)
3) Shell and utility facilities
4) Verification testing methods
5) Secure/Trusted system considerations
6) Network interface facilities
7) System administration
8) Graphical user interfaces
9) Profiles describing application- or user-specific combinations of open systems standards for: supercomputing, multiprocessor, and batch extensions; transaction processing; realtime systems; and multiuser systems based on historical models
10) An overall guide to POSIX-based or related open systems standards and profiles

Extensions are approved as “amendments” or “revisions” to this document, following IEEE Standards procedures.

Approved amendments are published separately until the full document is reprinted and such amendments are incorporated in their proper positions.

If you have an interest in participating in the PASC working groups addressing these issues, please send your name, address, and phone number to the Secretary, IEEE Standards Board, Institute of Electrical and Electronics Engineers, Inc., P.O. Box 1331, 445 Hoes Lane, Piscataway, NJ 08855-1331, and ask to have this forwarded to the chair of the appropriate PASC working group. If you have an interest in participating in this work at the international level, contact your ISO/IEC national body.

IEEE Std 1003.1-1990 was prepared by the IEEE P1003.1 working group, sponsored by the Technical Committee on Operating Systems and Application Environments of the IEEE Computer Society. At the time this standard was approved, the membership of the IEEE P1003.1 working group was as follows:

[6]A *Standards Status Report* that lists all current IEEE Computer Society standards projects is available from the IEEE Computer Society, 1730 Massachusetts Avenue NW, Washington, DC 20036-1903; Telephone: +1 202 371-0101; FAX: +1 202 728-9614.

Allen Hankinson
Steve Head
Barry Hedquist
William Henderson
Lee A. Hollaar
Terrence Holm
Randall Howard
Irene Hu
Andrew Huber
Richard Hughes-Rowlands
Judith Hurwitz
Jim Isaak
Dan Iuster
Richard James
Hal Jespersen
Michael J. Karels
Sol Kavy
Lorraine C. Kevra
Jeffrey S. Kimmel
M. J. Kirk
Dale Kirkland
John T. Kline
Kenneth Klingman
Joshua Knight
Andrew R. Knipp
David Korn
Don Kretsch
Takahiko Kuki
Thomas Kwan
Robin B. Lake
Mark Lamonds
Doris Lebovits
Maggie Lee
Greger Leijonhufvud
Robert Lenk
David Lennert
Donald Lewine
Kevin Lewis
F. C. Lim
James Lonjers
Warren E. Loper
Roger Martin
Martin J. McGowan
Marshall McKusick
Robert McWhirter
Paul Merry
Doug Michels
Gary W. Miller
James Moe
James W. Moore
Martha Nalebuff
Barry Needham
Alan F. Nugent
Jim Oldroyd
Craig Partridge
John Peace
John C. Penney
P. Plauger
Gerald Powell
Scott E. Preece
Joseph Ramus
Wendy Rauch
Carol Raye
Wayne B. Reed
Christopher J. Riddick
Andrew K. Roach
Robert Sarr
Lorne H. Schachter
Norman Schneidewind
Stephen Schwarm
Richard Scott
Leonard Seagren
Glen Seeds
Karen Sheaffer
Charles Smith
Steven Sommars
Douglas H. Steves
James Tanner
Ravi Tavakley
Marc Teitelbaum
Donn S. Terry
Gary F. Tom
Andrew Twigger
Mark-Rene Uchida
L. David Umbaugh
Michael W. Vannier
David John Wallace
Stephen Walli
Larry Wehr
Bruce Weiner
Robert Weissensee
P. J. Weyman
Andrew Wheeler, Jr.
David Willcox
Randall F. Wright
Oren Yuen
Jason Zions

IEEE Std 1003.1b-1993 was prepared by the IEEE P1003.4 working group, sponsored by the Portable Applications Standards Committee of the IEEE Computer Society. At the time this standard was approved, the membership of the IEEE P1003.4 working group was as follows:

Bernard Cox
Denise Cully
Charles Curley
Ajit Dandapani
David Dodge
Mark Doran
Terence Dowling
Larry Dwyer
Ron Elliott
Philip Erickson
Lemeul Eubanks
Fran Fadden
Allen Farris
Kester Fong
Martin Fouts
Dan Frank
Art Fritzson
Mitchell Fuchs
Bill Gallmeister
Bob Gambrel
John Gertwagen
Greg Goddard
W. E. Goebel
Andrew Gollan
Rup Grafendorfer
David Greenstein
Rick Greer
Jerry Gross
Dan Grostick
Mesut Gunduc
Bob Hairfield
John Hanley
Richard Hart
Terry Hayes
Stephen Head
Mats Hellstrom
Morris Herbert
Russ Holt
Ian Hopper
Jim Houston
John Howard
Bill Hullsiek
James Isaak
Doug Jewett
Mike Jones
Christopher Juillet
Vassilios Kalfakakos
Peter Kao
Tom Kapish
Lars Karlsson
Sol Kavy

Gregg Kellogg
Doug Kevorkian
Sandeep Khanna
Alan Kiecker
Jeff Kimmel
Dale Kirlkand
Y. K. Kiu
Bob Kleinschumidf
John Kline
Robert Knighten
Joseph Korty
K. Kothari
Eva Kristensson
Peter Krupp
Jeff Lee
Bob Lenk
Kin Leung
Doug Locke
Juliam Lomberg
Michel Lortie
Brian Lucas
Robert Luken
Dave Lunger
Ron Mabe
Rod MacDonald
Sukan Makmuri
Mark Manasse
Mike Manley
Steve Marcie
Mike Marciniszyn
Michael McBride
Brian McCarthy
Dora Merris
John Meyer
Laura Micks
Stephen Miller
Cliff Moore
Jim Moseman
Naren Nachiappan
Martha Nalebuff
Bret Needle
Mark Nudelman
Greg Nuss
Bob Nystrom
Robin O'Neal
Hagai Ohel
John Parker
Robert Parlock
Simon Patience
Offer Pazy
Donald Peterson

Prayoon Phathayakorn
Gilbert Pile Jr
Dave Plauger
Wendy Rauch-Hindin
Dave Rorke
Barry Ruzek
Doris Ryan
Ashok Saxena
Rich Schaaf
Norm Scherer
Lee Schermerhorn
Curt Schimmel
Mike Schultz
Ed Schwartz
Tom Scott
Karen Sheaffer
Thomas Shonk
Bruce Sigmon
Inder Singh
Roger Sippl
Val Skalabrin
Carl Smith
James Soddy
K. Y. Srinivasan
Daniel Steinberg
Audrey Strathmeyer
Keith Stuck
Garret Swart
Ravi Tavakley
Marc Teller
Jack Test
Dan Tiernan
Barry Traylor
Michael Turner
David Uhrlaub
Peter vanderLinden
Luis Varges
Joel Wagner
Alan Weaver
Perry Weller
Andrew Wheeler, Jr.
Gary Whisenhunt
Jack White
Dwight Wilcox
John Williams
John Wu
Margaret Yang
Tohru Yoneda
Steve Zanoni
Fred Zlotnick
John Zolnowsky

The following persons were members of the balloting group for IEEE Std 1003.1b-1993 :

Shane McCarron *Unix International Organizational Representative*

Miguel Abdo
Steven Albert
Bill Allen
Helene Armitage
David Athersych
Theodore P. Baker
Steven E. Barber
Robert Barned
John Barr
Edward Benson
Robert Bismuth
Nawaf Bitar
Jim Blondeau
James Bohem
Kathryn Bohrer
William Bone
Robert Borochoff
Keith Bostic
James P. Bound
Kevin Brady
Sven Brehmer
Charles Brooks
David Brower
Gretchen Brown
Ray Bryant
Luis-Felipe Cabrera
Nicholas A. Camillone
Clyde Camp
George S. Carson
Steven Carter
Jerry Cashin
John Caywood
Kilnam Chon
Philip Christopher
Anthony Cincotta
Robert L. Claeson
Mark Colburn
R. Cornelius
William M. Corwin
Mike Cossey
William Cox
Donald Cragun
Russell Davis
Ana Maria De Alvare
Virgil Decker
Dave Decot
Larry Diegel
David Dodge
Terence Dowling
Stephen A. Dum
John D. Earls
Ron Elliott
Richard W. Elwood
David Emery
Philip H. Enslow
Philip Erickson
Catherine Fitzpatrick
Kester Fong
Daniel M. Frank
Bill Gallmeister
Michel Gien
Jean Gilmore
Richard Greer
Tom Griest
Dave Grindeland
Robert C. Groman
Judy Guist
Carl Hall
Charles Hammons
Allen L. Hankinson
Michael J. Hannah
Carol J. Harkness
Craig Harmer
Steve Head
Myron Hecht
Barry Hedquist
William Hefley
Hans H. Heilborn
Ralph Henkaus
Karl Heubaum
Jim Hightower
David F. Hinnant
John Hogan
Terrence Holm
Rand Hoven
Randall Howard
Irene Hu
Andrew R. Huber
Richard Hughes-Rowlands
William Hullsiek
Judith Hurwitz
Jim Isaak
Dan Iuster
Hal Jespersen
Michael Jones
Greg Jones
James E. Jordan
Mike Kamrad
Michael J. Karels
Sol Kavy
Michael Kearney
Joseph Keenan
Jerry Keselman
Lorraine C. Kevra
Sandeep Khanna
Alan W. Kiecker
Jeffrey S. Kimmel
Martin J. Kirk
Dale Kirkland
Steve Kleiman
Philip Klimbal
John T. Kline
Kenneth Klingman
Robert Knighten
Takashi Kojo
David Korn
D. Richard Kuhn
Takahiko Kuki
Joan Kundig
Thomas Kwan
Robin B. Lake
Mike Lambert
Mark Lamonds
Sue Le Grand
Doris Lebovits
Maggie Lee
Greger Leijonhufvud
Robert Lenk
David Lennert
Donald Lewine
Kevin Lewis
Kin Fun Li
F. C. Lim
John Litke
C. Douglass Locke
James P. Lonjers
Warren E. Loper
Joseph F. P. Luhukay
Robert D. Luken
Dave Lunger
Heinz Lycklama

Also included are the following nonvoting IEEE Standards Board liaisons:

IEEE Std 1003.1c-1995 was prepared by the IEEE P1003.4 working group, sponsored by the Portable Applications Standards Committee of the IEEE Computer Society. At the time this standard was approved, the membership of the IEEE P1003.4 working group was as follows:

Tim Carter
Philip Christopher
Jeff Cline
Robert A. Conti
William Corwin
Mike R. Cossey
Bernard Cox
William Cox
Charles Curley
Denise Cully
Ajit Dandapani
Dave Decot
Steven R. Deller
Mark Doran
Dennis Dougherty
Terence S. Dowling
Maryland R. Edwards
David Emery
Philip J. Erickson
Lemeul Eubanks
Fran Fadden
Jesse Fang
Bengt Farre
Kester Fong
Martin Fouts
Daniel M. Frank
Bill Gallmeister
John A. Gertwagen
James Gibson
Greg Goddard
W. E. Goebel
Karen D. Gorden
Benny M. Gothard
Howard Green
Richard Greer
Jerry Gross
Joe Gwinn
Barbara Haleen
Geoff Hall
Stephen Head
Rick Holland
William Hullsiek
Michael B. Jones
Dan Juttelstad
Vassilios Kalfakakos
Peter Kao
Glenn Kasten
Sol Kavy
Sandeep Khanna
Alan W. Kiecker
Steven Kleiman
John T. Kline
Robert Knighten
Keith A. Kohout
Joseph A. Korty
K. Kothari
Eva Kristensson
Bernard Ku
Frank Lawlor
Michael Lazar
Eugene Leache
C. Douglass Locke
Robert D. Luken
Dave Lunger
Mark S. Manasse
Brian S. McCarthy
Roy McKean
Steve McPolin
John Meyer
Laura Micks
Hehdell Miller
Stephen Miller
Robert E. Murphy
Narendran Nachiappan
Martha Nalebuff
Bret Needle
Mei Ng
Mark Nudelman
Tricia Oberndorf
Jack O'Quin
Robert Parlock
Simon Patience
Offer Pazy
Prayoon Phathayakorn
Chris Phillips
Aiko Pinkoski
Dave Plauger
Igor Podboj
Charlas Prass
Franklin C. Prindle
Chris Quenelle
Wendy Rauch
Dave Rorke
Helmut Roth
Barry Ruzek
John Sauter
Richard Schaaf
Lee Schermerhorn
Mike Schultz
Tom Scott
Lui Sha
Devang K. Shah
Bruce Sigmon
Inder Singh
Charles R. Smith
Peter H. Smith
James Soddy
K. Y. Srinivasan
Audrey Strathmeyer
R. Stroobosscher
Keith Stuck
Del Swanson
Garret Swart
Ravi Tavakley
Stephen Thompson
Dan Tiernan
Barry Traylor
David R. Uhrlaub
Marl VandenBrink
Dave Vogel
Stephen R. Walli
Mary Weeks
Andrew E. Wheeler, Jr.
David Willcox
Dave Winchell
Tohru Yoneda
George Zerdian
Fred Zlotnick
John Zolnowsky

The following persons were members of the balloting group for IEEE Std 1003.1c-1995:

Paul Rabin	*Open Software Foundation Organizational Representative*
Richard Alexander	*Share, Incorporated Organizational Representative*
Robert Boucher	*Uniforum Organizational Representative*

Gary M. Allen
Helen Armitage
Charles R. Arnold
Bengt Asker
David Athersych
Randall Atkinson
Timothy Baker
Theodore P. Baker
Ralph Barker
John Barr
Edward Benson
Richard M. Bergman
Andy R. Bihain
Robert Bismuth
Nawaf Bitar
George Bittner
Win Bo
James Bohem
Kathryn Bohrer
Keith Bostic
Sven L. Brehmer
Dale Brouhard
David Brower
Gretchen Brown
David Butenhof
Steven Carter
Jerry Cashin
John Caywood
Siddhartha Chatterjee
Andy B. Cheese
Janice Chelini
James Chelini
Kilnam Chon
Robert L. Claeson
Robert A. Conti
Eric Cooper
William Corwin
Mike R. Cossey
William Cox
Donald Cragun
Ana Maria DeAlvare
Virgil Decker
Steven R. Deller
David Dodge
Terence S. Dowling
Richard P. Draves
Stephen A. Dum
John D. Earls
David A. Eckhardt
Edna B. Edelman
Ron Elliott
Philip H. Enslow
Philip J. Erickson
Kester Fong
Daniel M. Frank
Bill Gallmeister
Mitchell Gart
John A. Gertwagen
James Gibson
Michel Gien
Karen D. Gordon
Richard Greer
Tom Griest
Dave Grindeland
Judy Guist
Gregory Guthrie
Mark D. Guzzi
Joe Gwinn
Barbara Haleen
Charles E. Hammons
Charles Harkey
Carol J. Harkness
Myron Hecht
Hans H. Heilborn
Karl Heubaum
David F. Hinnant
Rand Hoven
Steven Howell
Irene Hu
Hai Huang
Andrew R. Huber
Richard Hughes-Rowlands
William Hullsiek
Aron K. Insinga
Jim Isaak
Richard E. James
Hal Jespersen
Lowell Johnson
Michael B. Jones
James E. Jordan
Daniel P. Julin
Mike Kamrad
Michael J. Karels
Michael Kearney
Lawrence J. Kenah
Judy Kerner
Lorraine C. Kevra
Sandeep Khanna
Jeffrey S. Kimmel
Paul J. King
Martin J. Kirk
David Kirshen
Richard J. C. Kissel
Steven Kleiman
Philip Klimbal
John T. Kline
Kenneth Klingman
Robert Knighten
Jens Kolind
Ronnie Kon
D. Richard Kuhn
Doris Lebovits
Robert Lenk
David Lennert
Donald Lewine
Kevin Lewis
Fang Ching Lira
John Litke
C. Douglass Locke
Warren E. Loper
Robert D. Luken
Robert Makowski
William Mar
Roger Martin
Joberto S. B. Martins
Marshall Kirk McKusick
Robert McWhirter
Gary W. Miller
James Moe
James W. Moore
Narendran Nachiappan
Martha Nalebuff
Barry Needham
Daniel Nissen
Landon Curt Noll
Fred Noz
Simon Patience
Offer Pazy
Crispin Perdue
Pat Philip
Chris Phillips
Dave Plauger
Michael L. Powell
Scott E. Preece
Franklin C. Prindle
John S. Quarterman
Carol Raye
Paul E. Renaud
Christopher J. Riddick
Andrew K. Roach
David Rorke
Seth Rosenthal
Helmut Roth
Paul Roy
John Sauter
Richard Schaaf
Lorne H. Schachter
Carl Schmiedekamp
Richard L. Scott
Leonard W. Seagren
Glen Seeds
Richard Seibel

Devang K. Shah	Ravi Tavakley	Larry A. Wehr
Edward J. Sharpe	Donn S. Terry	Bruce Weiner
Dan Shia	Gary F. Tom	Andrew E. Wheeler, Jr.
Eric R. Shienbrood	Barry Traylor	Nicholos Ray Wilkes
Rajiv Sinha	Mark-Rene Uchida	David Willcox
Jacob Slonim	Steven M. Valentine	John R. Williams
Charles R. Smith	James Van Sciver	David Wong
Richard Stallman	Michael W. Vannier	Tohru Yoneda
Jacqueline Stewart	Dalibor F. Vrsalovic	Oren Yuen
Peter D. Stout	Stephen R. Walli	Janusz Zalewski
Keith Stuck	Tom Watson	Jason Zions
Scott Sutter	Alan Weaver	John Zolnowsky
Del Swanson	Mary Weeks	

IEEE Std 1003.1i-1995 was prepared by the IEEE P1003.4 working group, sponsored by the Portable Applications Standards Committee of the IEEE Computer Society. At the time this standard was approved, the membership of the IEEE P1003.4 working group was as follows:

IEEE Std 1003.1-1990 was approved by the American National Standards Institute (ANSI) on April 15, 1991. IEEE Std 1003.1b-1993 was approved by ANSI on April 14, 1994. IEEE Std 1003.1c-1995 was approved by ANSI on June 7, 1996. IEEE Std 1003.1i-1995 was approved by ANSI on January 12, 1996.

CLAUSE PAGE

Information technology—Portable Operating System Interface (POSIX®)—Part 1: System Application Program Interface (API) [C Language]

1. General

1.1 Scope

This part of ISO/IEC 9945 defines a standard operating system interface and environment to support application portability at the source-code level. It is intended to be used by both application developers and system implementors.

This part of ISO/IEC 9945 comprises four major components:

1) Terminology, concepts, and definitions and specifications that govern structures, headers, environment variables, and related requirements
2) Definitions for system service interfaces and subroutines
3) Language-specific system services for the C programming language
4) Interface issues, including portability, error handling, and error recovery

The following areas are outside of the scope of this part of ISO/IEC 9945:

1) User interface (shell) and associated commands
2) Networking protocols and system call interfaces to those protocols
3) Graphics interfaces
4) Database management system interfaces
5) Record I/O considerations
6) Object or binary code portability
7) System configuration and resource availability

This part of ISO/IEC 9945 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 and applications with realtime requirements. The interfaces included were the set required to make this part of ISO/IEC 9945 minimally usable to realtime applications on single processor systems. The definition of *realtime* used in defining the scope of this standard is:

> “Realtime in operating systems: the ability of the operating system to provide a required level of service in a bounded response time.”

This standard includes system interfaces to support applications with requirements for multiple flows of control, called *threads*, within a process. The facilities provided support a convenient interface for writing multithreaded applications.

The key elements defining the scope are

1) Defining a sufficient set of functionality to cover a significant part of the realtime application program domain
2) Defining sufficient performance constraints and performance-related functions to allow a realtime application to achieve deterministic response from the system
3) Defining a sufficient set of functionality to support multiple threads of control within a process

Specifically within the scope is to define interfaces that do not preclude high-performance implementations on traditional uniprocessor realtime systems. The interfaces in this standard are specifically targeted at supporting tightly coupled multitasking environments, including multiprocessors and advanced language constructs. Wherever possible, the requirements of other application environments are included in this interface definition. It is beyond the scope of these interfaces to support networking functionality.

This part of ISO/IEC 9945 has been defined exclusively at the source-code level. The objective is that a Strictly Conforming POSIX.1 Application source program can be translated to execute on a conforming implementation. Additionally, although the interfaces will be portable, some of the numeric parameters used by an implementation may have hardware dependencies.

1.2 Normative References

The following standards contain provisions which, through references in this text, constitute provisions of this part of ISO/IEC 9945. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this part of ISO/IEC 9945 are encouraged to investigate the possibility of applying the most recent editions of the standards listed below. Members of IEC and ISO maintain registers of currently valid International Standards.

{1} ISO/IEC 646 : 1991,[1] *Information processing—ISO 7-bit coded character set for information interchange.*

{2} ISO/IEC 9899 : 1990, *Programming languages—C.*

1.3 Conformance

1.3.1 Implementation Conformance

1.3.1.1 Requirements

A *conforming implementation* shall meet all of the following criteria:

[1]ISO/IEC documents can be obtained from the ISO office, 1, rue de Varembé, Case Postale 56, CH-1211, Genève 20, Switzerland/Suisse.

1) The system shall support all required interfaces defined within this part of ISO/IEC 9945 that are not described under the {_POSIX_THREADS} option. If {_POSIX_THREADS} is defined, then the system shall also support all required interfaces described under the {_POSIX_THREADS} option. All supported interfaces shall support the functional behavior described herein.
2) The system may provide additional functions or facilities not required by this part of ISO/IEC 9945. Nonstandard extensions of the functions or facilities specified in this part of ISO/IEC 9945 should be identified as such in the system documentation. Nonstandard extensions, when used, may change the behavior of functions or facilities defined by this part of ISO/IEC 9945. The conformance document shall define an environment in which an application can be run with the behavior specified by the standard. In no case shall such an environment require modification of a Strictly Conforming POSIX.1 Application.

1.3.1.2 Documentation

A conformance document with the following information shall be available for an implementation claiming conformance to this part of ISO/IEC 9945. The conformance document shall have the same structure as this part of ISO/IEC 9945, with the information presented in the appropriately numbered sections, clauses, and subclauses. The conformance document shall not contain information about extended facilities or capabilities outside the scope of this part of ISO/IEC 9945.

The conformance document shall contain a statement that indicates the full name, number, and date of the standard that applies. The conformance document may also list international software standards that are available for use by a Conforming POSIX.1 Application. Applicable characteristics where documentation is required by one of these standards, or by standards of government bodies, may also be included.

The conformance document shall describe the limit values found in the `<limits.h>` and `<unistd.h>` headers, stating values, the conditions under which those values may change, and the limits of such variations, if any.

The conformance document shall describe the behavior of the implementation for all implementation-defined features defined in this part of ISO/IEC 9945. This requirement shall be met by listing these features and providing either a specific reference to the system documentation or providing full syntax and semantics of these features. The conformance document may specify the behavior of the implementation for those features where this part of ISO/IEC 9945 states that implementations may vary or where features are identified as undefined or unspecified.

No specifications other than those described in this part of ISO/IEC 9945 shall be present in the conformance document.

The phrases “shall document” or “shall be documented” in this part of ISO/IEC 9945 mean that documentation of the feature shall appear in the conformance document, as described previously, unless the system documentation is explicitly mentioned.

The system documentation should also contain the information found in the conformance document.

1.3.1.3 Conforming Implementation Options

The following symbolic constants, described in the subclauses indicated, reflect implementation options for this part of ISO/IEC 9945 that could warrant requirement by Conforming POSIX.1 Applications, or in specifications of conforming systems, or both:

{NGROUPS_MAX}	Supplementary Groups option (in 2.8.3)
{_POSIX_ASYNCHRONOUS_IO}	Asynchronous Input and Output option (in 2.9.3)
{_POSIX_CHOWN_RESTRICTED}	Change File Owner Restriction (in 2.9.4)
{_POSIX_FSYNC}	File Synchronization option (in 2.9.3)

{_POSIX_JOB_CONTROL}	Job Control option (in 2.9.3)
{_POSIX_MAPPED_FILES}	Memory Mapped Files option (in 2.9.3)
{_POSIX_MEMLOCK}	Process Memory Locking option (in 2.9.3)
{_POSIX_MEMLOCK_RANGE}	Range Memory Locking option (in 2.9.3)
{_POSIX_MEMORY_PROTECTION}	Memory Protection option (in 2.9.3)
{_POSIX_MESSAGE_PASSING}	Message Passing option (in 2.9.3)
{_POSIX_PRIORITIZED_IO}	Prioritized Input and Output option (in 2.9.3)
{_POSIX_PRIORITY_SCHEDULING}	Process Scheduling option (in 2.9.3)
{_POSIX_REALTIME_SIGNALS}	Realtime Signals Extension (in 2.9.3)
{_POSIX_SEMAPHORES}	Semaphores option (in 2.9.3)
{_POSIX_SHARED_MEMORY_OBJECTS}	Shared Memory Objects option (in 2.9.3)
{_POSIX_SYNCHRONIZED_IO}	Synchronized Input and Output option (in 2.9.3)
{_POSIX_TIMERS}	Timers option (in 2.9.3)
{_POSIX_THREAD_PRIO_INHERIT}	Priority Inheritance option (in 2.9.3)
{_POSIX_THREAD_PRIORITY_SCHEDULING}	Thread Execution Scheduling option (in 2.9.3)
{_POSIX_THREADS}	Threads option (in 2.9.3)
{_POSIX_THREAD_SAFE_FUNCTIONS}	Thread-Safe Functions option (in 2.9.3)

The remaining symbolic constants in 2.9.3 and 2.9.4 are useful for testing purposes and as a guide to applications on the types of behaviors they need to be able to accommodate. They do not reflect sufficient functional difference to warrant requirement by Conforming POSIX. 1 Applications or to distinguish between conforming implementations.

In the cases where omission of an option would cause functions described by this part of ISO/IEC 9945 to not be defined, an implementation shall provide a function that is callable with the syntax defined in this part of ISO/IEC 9945, even though in an instance of the implementation the function may always do nothing but return an error.

1.3.2 Application Conformance

All applications claiming conformance to this part of ISO/IEC 9945 shall use only language-dependent services for the C programming language described in 1.3.3 and shall fall within one of the following categories:

1.3.2.1 Strictly Conforming POSIX.1 Application

A Strictly Conforming POSIX.1 Application is an application that requires only the facilities described in this part of ISO/IEC 9945 and the applicable language standards. Such an application shall accept any behavior described in this part of ISO/IEC 9945 as *unspecified* or *implementation-defined*, and for symbolic constants, shall accept any value in the range permitted by this part of ISO/IEC 9945. Such applications are permitted to adapt to the availability of facilities whose availability is indicated by the constants in 2.8.2 and 2.9.

1.3.2.2 Conforming POSIX.1 Application

1.3.2.2.1 ISO/IEC Conforming POSIX.1 Application

An ISO/IEC Conforming POSIX.1 Application is an application that uses only the facilities described in this part of ISO/IEC 9945 and approved Conforming Language bindings for any ISO or IEC standard. Such an application shall include a statement of conformance that documents all options and limit dependencies, and all other ISO or IEC standards used.

1.3.2.2.2 <National Body> Conforming POSIX.1 Application

A <National Body> Conforming POSIX.1 Application differs from an ISO/IEC Conforming POSIX.1 Application in that it also may use specific standards of a single ISO/IEC member body referred to here as "<*National Body*>." Such an application shall include a statement of conformance that documents all options and limit dependencies, and all other <National Body> standards used.

1.3.2.3 Conforming POSIX.1 Application Using Extensions

A Conforming POSIX.1 Application Using Extensions is an application that differs from a Conforming POSIX.1 Application only in that it uses nonstandard facilities that are consistent with this part of ISO/IEC 9945. Such an application shall fully document its requirements for these extended facilities, in addition to the documentation required of a Conforming POSIX.1 Application. A Conforming POSIX.1 Application Using Extensions shall be either an ISO/IEC Conforming POSIX.1 Application Using Extensions or a <National Body> Conforming POSIX.1 Application Using Extensions (see 1.3.2.2.1 and 1.3.2.2.2).

1.3.3 Language-Dependent Services for the C Programming Language

Parts of ISO/IEC 9899 {2} (hereinafter referred to as the "C Standard {2}") will be referenced to describe requirements also mandated by this part of ISO/IEC 9945. The sections of the C Standard {2} referenced to describe requirements for this part of ISO/IEC 9945 are specified in Section 8 Section 8 also sets forth additions and amplifications to the referenced sections of the C Standard {2}. Any implementation claiming conformance to this part of ISO/IEC 9945 with the C Language Binding shall provide the facilities referenced in Section 8, along with any additions and amplifications Section 8 requires.

Although this part of ISO/IEC 9945 references parts of the C Standard {2} to describe some of its own requirements, conformance to the C Standard {2} is unnecessary for conformance to this part of ISO/IEC 9945. Any C language implementation providing the facilities stipulated in Section 8 may claim conformance; however, it shall clearly state that its C language does not conform to the C Standard {2}.

1.3.3.1 Types of Conformance

Implementations claiming conformance to this part of ISO/IEC 9945 with the C Language Binding shall claim one of two types of conformance—conformance to POSIX.1, C Language Binding (C Standard Language-Dependent System Support), or to POSIX.1, C Language Binding (Common-Usage C Language-Dependent System Support).

1.3.3.2 C Standard Language-Dependent System Support

Implementors shall meet the requirements of Section 8 using for reference the C Standard {2}. Implementors shall clearly document the version of the C Standard {2} referenced in fulfilling the requirements of Section 8

Implementors seeking to claim conformance using the C Standard {2} shall claim conformance to POSIX.1, C Language Binding (C Standard Language-Dependent System Support).

1.3.3.3 Common-Usage C Language-Dependent System Support

Implementors, instead of referencing the C Standard {2}, shall provide the tines and support required in Section 8 using common usage as guidance. Implementors shall meet all the requirements of Section 8 except where references are made to the C Standard {2}. In places where the C Standard {2} is referenced, implementors shall provide equivalent support in a manner consistent with common usage of the C programming language. Implementors shall document, in Section 8 of the conformance document, all differences between the interface provided and the interface that would have been provided had the C Standard {2} been implemented instead of common usage. Implementors shall clearly document the version of the C Standard {2} referenced in documenting interface differences and should issue updates on differences for all new versions of the C Standard {2}.

Where a function has been introduced by the C Standard {2}, and thus there is no common-usage referent for it, if the function is implemented, it shall be implemented as described in the C Standard {2}. If the function is not implemented, it shall be documented as a difference from the C Standard {2} as required above.

1.3.4 Other C Language-Related Specifications

The following rules apply to the usage of C language library functions; each of the statements in this subclause applies to the detailed function descriptions in Sections 3 through 9 and 11 through 18, unless explicitly stated otherwise:

1) If an argument to a function has an invalid value (such as a value outside the domain of the function, or a pointer outside the address space of the program, or a **NULL** pointer when that is not explicitly permitted), the behavior is undefined.
2) Any function may also be implemented as a macro in a header. Applications should use #undef to remove any macro definition and ensure that an actual function is referenced. Applications should also use #undef prior to declaring any function in this part of ISO/IEC 9945.
3) Any invocation of a library function that is implemented as a macro shall expand to code that evaluates each of its arguments only once, fully protected by parentheses where necessary, so it is generally safe to use arbitrary expressions as arguments.
4) Provided that a library function can be declared without reference to any type defined in a header, it is also permissible to declare the function, either explicitly or implicitly, and use it without including its associated header.
5) If a function that accepts a variable number of arguments is not declared (explicitly or by including its associated header), the behavior is undefined.

1.3.5 Other Language-Related Specifications

This part of ISO/IEC 9945 is currently specified in terms of the language defined by the C Standard {2}. Bindings to other programming languages are being developed.

If conformance to this part of ISO/IEC 9945 is claimed for implementation of any programming language, the implementation of that language shall support the use of external symbols distinct to at least 31 bytes in length in the source program text. (That is, identifiers that differ at or before the thirty-first byte shall be distinct.) If a national or international standard governing a language defines a maximum length that is less than this value, the language-defined maximum shall be supported. External symbols that differ only by case shall be distinct when the character set in use distinguishes upper- and lowercase characters and the language permits (or requires) upper- and lowercase characters to be distinct in external symbols.

Subsequent sections of this part of ISO/IEC 9945 refer only to the C Language.

2. Terminology and General Requirements

2.1 Conventions

This part of ISO/IEC 9945 uses the following typographic conventions:

1) The *italic* font is used for:
   - Cross references to defined terms within 1.3, 2.2.1, and 2.2.2; symbolic parameters that are generally substituted with real values by the application
   - C language data types and function names (except in function Synopsis subclauses)
   - Global external variable names
   - Function families; references to groups of closely related functions (such as *directory*, *exec*, *sigsetops*, *sigwait*, *stdio*, and *wait*)
2) The **bold** font is used with a word in all capital letters, such as

   PATH

   to represent an environment variable, as described in 2.6. It is also used for the term "**NULL** pointer."
3) The `constant-width` (Courier) font is used:
   - For C language data types and function names within function Synopsis subclauses
   - To illustrate examples of system input or output where exact usage is depicted
   - For references to utility names and C language headers
   - For names of attributes in attributes objects
4) Symbolic constants returned by many functions as error numbers are represented as:

   [ERRNO]

   See 2.4.
5) Symbolic constants or limits defined in certain headers are represented as:

   {LIMIT}

   See 2.8 and 2.9.

In some cases tabular information is presented "inline"; in others it is presented in a separately labeled table. This arrangement was employed purely for ease of typesetting and there is no normative difference between these two cases.

The conventions listed previously are for ease of reading only. Editorial inconsistencies in the use of typography are unintentional and have no normative meaning in this part of ISO/IEC 9945.

NOTEs provided as parts of labeled tables and figures are integral parts of this part of ISO/IEC 9945 (normative). Footnotes and notes within the body of the text are for information only (informative).

Numerical quantities are presented in international style: comma is used as a decimal sign and units are from the International System (SI).

2.2 Definitions

2.2.1 Terminology

For the purposes of this part of ISO/IEC 9945, the following definitions apply:

2.2.1.1 conformance document: A document provided by an implementor that contains implementation details as described in 1.3.1.2.

2.2.1.2 implementation defined: An indication that the implementation shall define and document the requirements for correct program constructs and correct data of a value or behavior.

2.2.1.3 may: An indication of an optional feature.

With respect to implementations, the word *may* is to be interpreted as an optional feature that is not required in this part of ISO/IEC 9945, but can be provided. With respect to Strictly Conforming POSIX.1 Applications, the word *may* means that the optional feature shall not be used.

2.2.1.4 obsolescent: An indication that a certain feature may be considered for withdrawal in future revisions of this part of ISO/IEC 9945.

Obsolescent features are retained in this version because of their widespread use. Their use in new applications is discouraged.

2.2.1.5 shall: An indication of a requirement on the implementation or on Strictly Conforming POSIX.1 Applications, where appropriate.

2.2.1.6 should:

1) With respect to implementations, an indication of an implementation recommendation, but not a requirement.
2) With respect to applications, an indication of a recommended programming practice for applications and a requirement for Strictly Conforming POSIX.1 Applications.

2.2.1.7 supported: A condition regarding optional functionality.

Certain functionality in this part of ISO/IEC 9945 is optional, but the interfaces to that functionality are always required. If the functionality is *supported*, the interfaces work as specified by this part of ISO/IEC 9945 (except that they do not return the error condition indicated for the unsupported case). If the functionality is not *supported*, the interface shall always return the indication specified for this situation.

2.2.1.8 system documentation: All documentation provided with an implementation, except the conformance document.

Electronically distributed documents for an implementation are considered part of the system documentation.

2.2.1.9 undefined: An indication that this part of ISO/IEC 9945 imposes no portability requirements on an application's use of an indeterminate value or its behavior with erroneous program constructs or erroneous data.

Implementations (or other standards) may specify the result of using that value or causing that behavior. An application using such behaviors is using extensions, as defined in 1.3.2.3.

2.2.1.10 unspecified: An indication that this part of ISO/IEC 9945 imposes no portability requirements on applications for correct program constructs or correct data regarding a value or behavior.

Implementations (or other standards) may specify the result of using that value or causing that behavior. An application requiring a specific behavior, rather than tolerating any behavior when using that functionality, is using extensions, as defined in 1.3.2.3.

2.2.2 General Terms

For the purposes of this part of ISO/IEC 9945 , the following definitions apply:

2.2.2.1 absolute pathname: See pathname resolution in 2.3.6.

2.2.2.2 access mode: A form of access permitted to a file.

2.2.2.3 address space: The memory locations that can be referenced by the threads of a process.

2.2.2.4 appropriate privileges: An implementation-defined means of associating privileges with a implementation defined process with regard to the function calls and function call options defined in this part of ISO/IEC 9945 that need special privileges.

There may be zero or more such means.

2.2.2.5 arm (a timer): To start a timer measuring the passage of time, enabling notifying a process when the specified time or time interval has passed.

2.2.2.6 async-cancel-safe function: A function that may be safely invoked by an application while the asynchronous form of cancellation is enabled.

No function in this standard is async-cancel safe unless explicitly described as such.

NOTE — See Section 18 for further clarifications of the meaning of this term.

2.2.2.7 async-signal-safe function: A function that may be invoked, without restriction, from signal-catching functions. (See 3.3.1.3.)

No function in this standard is async-signal safe unless explicitly described as such.

2.2.2.8 asynchronous I/O operation: An I/O operation that does not of itself cause the thread requesting the I/O to be blocked from further use of the processor.

This implies that the thread and the I/O operation may be running concurrently.

2.2.2.9 asynchronous I/O completion: For an asynchronous read or write operation, when a corresponding synchronous read or write would have completed and when any associated status fields have been updated.

2.2.2.10 asynchronously generated signal: A signal that is not attributable to a specific thread.

NOTE — Examples are: signals sent via *kill*( ), signals sent from the keyboard, and signals delivered to process groups. Being asynchronous is a property of how the signal was generated and not a property of the signal number. All signals may be generated asynchronously.

2.2.2.11 background process: A process that is a member of a background process group.

2.2.2.12 background process group: Any process group, other than a foreground process group, that is a member of a session that has established a connection with a controlling terminal.

2.2.2.13 block special file: A file that refers to a device.

A block special file is normally distinguished from a character special file by providing access to the device in a manner such that the hardware characteristics of the device are not visible.

2.2.2.14 blocked thread: A thread that is waiting for some condition (other than the availability of a processor) to be satisfied before it can continue execution.

2.2.2.15 character: A sequence of one or more bytes representing a single graphic symbol.

NOTE — This term corresponds in the C Standard {2} to the term *multibyte character*, noting that a single-byte character is a special case of multibyte character. Unlike the usage in the C Standard {2}, *character* here has no necessary relationship with storage space, and *byte* is used when storage space is discussed.

2.2.2.16 character special file: A file that refers to a device.

One specific type of character special file is a terminal device file, whose access is defined in 7.1. Other character special files have no structure defined by this part of ISO/IEC 9945, and their use is unspecified by this part of ISO/IEC 9945.

2.2.2.17 child process: See *process* in 2.2.2.88.

2.2.2.18 clock: An object that measures the passage of time.

The current value of the time measured by a clock can be queried and, possibly, set to a value within the legal range of the clock.

2.2.2.19 clock tick: An interval of time.

A number of these occur each second. Clock ticks are one of the units that may be used to express a value found in type *clock_t*.

2.2.2.20 condition variable: A synchronization object that allows a thread to suspend execution, repeatedly, until some associated predicate becomes true.

A thread whose execution is suspended on a condition variable is said to be *blocked on* the condition variable.

2.2.2.21 controlling process: The session leader that established the connection to the controlling terminal.

Should the terminal subsequently cease to be a controlling terminal for this session, the session leader shall cease to be the controlling process.

2.2.2.22 controlling terminal: A terminal that is associated with a session.

Each session may have at most one controlling terminal associated with it, and a controlling terminal is associated with exactly one session. Certain input sequences from the controlling terminal (see 7.1) cause signals to be sent to all processes in the process group associated with the controlling terminal.

2.2.2.23 current working directory: See working directory in 2.2.2.145.

2.2.2.24 device: A computer peripheral or an object that appears to the application as such.

2.2.2.25 directory: A file that contains directory entries.

No two directory entries in the same directory shall have the same name.

2.2.2.26 directory entry [link]: An object that associates a filename with a file. Several directory entries can associate names with the same file.

2.2.2.27 direct I/O: An operation that attempts to circumvent a system performance optimization for the optimization of the individual I/O operation.

2.2.2.28 disarm (a timer): To stop a timer from measuring the passage of time, disabling any future process notifications (until the timer is armed again).

2.2.2.29 drift rate (of a clock): The rate at which the time measured by a clock deviates from the actual passage of real time.

A positive drift rate causes a clock to gain time with respect to real time; a negative drift rate causes a clock to lose time with respect to real time.

2.2.2.30 dot: The filename consisting of a single dot character (.).

See pathname resolution in 2.3.6.

2.2.2.31 dot-dot: The filename consisting solely of two dot characters (..).

See pathname resolution in 2.3.6.

2.2.2.32 effective group ID: An attribute of a process that is used in determining various permissions, including file access permissions, described in B.2.3.2.

See *group ID*. This value is subject to change during the process lifetime, as described in 3.1.2 and 4.2.2.

2.2.2.33 effective user ID: An attribute of a process that is used in determining various permissions, including file access permissions.

See user ID. This value is subject to change during the process lifetime, as described in 3.1.2 and 4.2.2.

2.2.2.34 empty directory: A directory that contains, at most, directory entries for dot and dot-dot.

2.2.2.35 empty string [null string]: A character array whose first element is a null character.

2.2.2.36 Epoch: The time 0 hours, 0 minutes, 0 seconds, January 1, 1970, Coordinated Universal Time.

See seconds since the Epoch.

2.2.2.37 feature test macro: A #defined symbol used to determine whether a particular set of features will be included from a header.

See 2.7.1.

2.2.2.38 FIFO special file [FIFO]: A type of file with the property that data written to such a file is read on a first-in-first-out basis.

Other characteristics of FIFOs are described in 5.3.1, 6.4.1, 6.4.2, and 6.5.3.

2.2.2.39 file: An object that can be written to, or read from, or both.

A file has certain attributes, including access permissions and type. File types include regular file, character special file, block special file, FIFO special file, and directory. Other types of files may be defined by the implementation.

2.2.2.40 file description: See *open file description* in 2.2.2.71.

2.2.2.41 file descriptor: A per-process unique, nonnegative integer used to identify an open file for the purpose of file access.

2.2.2.42 file group class: The property of a file indicating access permissions for a process related to the process's group identification.

A process is in the file group class of a file if the process is not in the file owner class and if the effective group ID or one of the supplementary groups of the process matches the group ID associated with the file. Other members of the class may be implementation defined.

2.2.2.43 file mode: An object containing the file permission bits and other characteristics of a file, as described in 5.6.1.

2.2.2.44 filename: A name consisting of 1 to {NAME_MAX} bytes used to name a file.

The characters composing the name may be selected from the set of all character values excluding the slash character and the null character. The filenames dot and dot-dot have special meaning; see pathname resolution in 2.3.6. A filename is sometimes referred to as a pathname component.

2.2.2.45 file offset: The byte position in the file where the next I/O operation begins.

Each open file description associated with a regular file, block special file, or directory has a file offset. A character special file that does not refer to a terminal device may have a file offset. There is no file offset specified for a pipe or FIFO.

2.2.2.46 file other class: The property of a file indicating access permissions for a process related to the process's user and group identification.

A process is in the file other class of a file if the process is not in the file owner class or file group class.

2.2.2.47 file owner class: The property of a file indicating access permissions for a process related to the process's user identification.

A process is in the file owner class of a file if the effective user ID of the process matches the user ID of the file.

2.2.2.48 file permission bits: Information about a file that is used, along with other information, to determine if a process has read, write, or execute/search permission to a file.

The bits are divided into three parts: owner, group, and other. Each part is used with the corresponding file class of processes. These bits are contained in the file mode, as described in 5.6.1. The detailed usage of the file permission bits in access decisions is described in file access permissions in B.2.3.2.

2.2.2.49 file serial number: A per-file system unique identifier for a file.

File serial numbers are unique throughout a file system.

2.2.2.50 file system: A collection of files and certain of their attributes.

It provides a name space for file serial numbers referring to those files.

2.2.2.51 first open (of a file): When a process opens a file that is not currently an open file within any process.

2.2.2.52 foreground process: A process that is a member of a foreground process group.

2.2.2.53 foreground process group: A process group whose member processes have certain privileges, denied to processes in background process groups, when accessing their controlling terminal.

Each session that has established a connection with a controlling terminal has exactly one process group of the session as the foreground process group of that controlling terminal. See 7.1.1.4.

2.2.2.54 foreground process group ID: The process group ID of the foreground process group.

2.2.2.55 group ID: A nonnegative integer, which can be contained in an object of type *gid_t*, that is used to identify a group of system users.

Each system user is a member of at least one group. When the identity of a group is associated with a process, a group ID value is referred to as a real group ID, an effective group ID, one of the (optional) supplementary group IDs, or an (optional) saved set-group-ID.

2.2.2.56 job control: A facility that allows users to selectively stop (suspend) the execution of processes and continue (resume) their execution at a later point.

The user typically employs this facility via the interactive interface jointly supplied by the terminal I/O driver and a command interpreter. Conforming implementations may optionally support job control facilities; the presence of this option is indicated to the application at compile time or run time by the definition of the {_POSIX_JOB_CONTROL} symbol; see 2.9.

2.2.2.57 last close (of a file): When a process closes a file, resulting in the file not being an open file within any process.

2.2.2.58 link: See directory entry in 2.2.2.26.

2.2.2.59 link count: The number of directory entries that refer to a particular file.

2.2.2.60 login: The unspecified activity by which a user gains access to the system.

Each login shall be associated with exactly one login name.

2.2.2.61 login name: A user name that is associated with a login.

2.2.2.62 map: To create an association between a page-aligned range of the address space of a process and a range of physical memory or some memory object, such that a reference to an address in that range of the address space results in a reference to the associated physical memory or memory object.

The mapped memory or memory object is not necessarily memory-resident.

2.2.2.63 memory object: Either a file or shared memory object.

When used in conjunction with *mmap*(), a memory object will appear in the address space of the calling process.

2.2.2.64 memory-resident: Managed by the implementation in such a way as to provide an upper bound on memory access times.

2.2.2.65 message: Information that can be transferred among processes or threads by being added to and removed from a message queue.

A message consists of a fixed-size message buffer.

2.2.2.66 message queue: An object to which messages can be added and removed.

Messages may be removed in the order in which they were added or in priority order. Characteristics and interfaces associated with message queues are defined in Section 15

2.2.2.67 mode: A collection of attributes that specifies a file's type and its access permissions.

See file access permissions in B.2.3.2.

2.2.2.68 mutex: A synchronization object used to allow multiple threads to serialize their access to shared data.

This term is derived from the capability it provides, namely, mutual exclusion. The thread that has locked a mutex becomes its owner and remains the owner until that same thread unlocks the mutex.

2.2.2.69 null string: See empty string in 2.2.2.35.

2.2.2.70 open file: A file that is currently associated with a file descriptor.

2.2.2.71 open file description: A record of how a process or group of processes are accessing a file.

Each file descriptor shall refer to exactly one open file description, but an open file description may be referred to by more than one file descriptor. A file offset, file status (see Table 6.5), and file access modes (see Table 6.6) are attributes of an open file description.

2.2.2.72 orphaned process group: A process group in which the parent of every member is either itself a member of the group or is not a member of the group's session.

2.2.2.73 page: The granularity of process memory mapping or locking.

Physical memory and memory objects can be mapped into the address space of a process on page boundaries and in integral multiples of pages. Process address space can be locked into memory—made memory-resident—on page boundaries and in integral multiples of pages.

2.2.2.74 parent directory:

1) When discussing a given directory, the directory that both contains a directory entry for the given directory and is represented by the pathname dot-dot in the given directory.
2) When discussing other types of files, a directory containing a directory entry for the file under discussion.

This concept does not apply to dot and dot-dot.

2.2.2.75 parent process: See *process* in 2.2.2.88.

2.2.2.76 parent process ID: An attribute of a new process after it is created by a currently active process.

The parent process ID of a process is the process ID of its creator, for the lifetime of the creator. After the creator's lifetime has ended, the parent process ID is the process ID of an implementation-defined system process.

2.2.2.77 path prefix: A pathname, with an optional ending slash, that refers to a directory.

2.2.2.78 pathname: A string that is used to identify a file.

A pathname consists of, at most, {PATH_MAX} bytes, including the terminating null character. It has an optional beginning slash, followed by zero or more filenames separated by slashes. If the pathname refers to a directory, it may also have one or more trailing slashes. Multiple successive slashes are considered to be the same as one slash. A pathname that begins with two successive slashes may be interpreted in an implementation-defined manner, although more than two leading slashes shall be treated as a single slash. The interpretation of the pathname is described in 2.3.6.

2.2.2.79 pathname component: See filename in 2.2.2.44.

2.2.2.80 persistence: A mode for semaphores, shared memory, and message queues requiring that the object and its state (including data, if any) are preserved after the object is no longer referenced by any process.

Persistence of an object does not imply that the state of the object is maintained across a system crash or a system reboot.

2.2.2.81 pipe: An object accessed by one of the pair of file descriptors created by the *pipe*() function.

Once created, the file descriptors can be used to manipulate it, and it behaves identically to a FIFO special file when accessed in this way. It has no name in the file hierarchy.

2.2.2.82 portable filename character set: The set of characters from which portable filenames are constructed.

For a filename to be portable across conforming implementations of this part of ISO/IEC 9945 , it shall consist only of the following characters:

```
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
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
0 1 2 3 4 5 6 7 8 9 . _ -
```

The last three characters are the period, underscore, and hyphen characters, respectively. The hyphen shall not be used as the first character of a portable filename. Upper- and lowercase letters shall retain their unique identities between conforming implementations. In the case of a portable pathname, the slash character may also be used.

2.2.2.83 preallocation: The reservation of resources in a system for a particular use.

Preallocation does not imply that the resources are immediately allocated to that use, but merely indicates that they are guaranteed to be available in bounded time when needed.

2.2.2.84 preempted thread: A running thread whose execution is suspended due to another thread becoming runnable at a higher priority.

2.2.2.85 priority: A nonnegative integer associated with processes or threads, whose value is constrained to a range defined by the applicable scheduling policy.

Numerically higher values represent higher priorities.

2.2.2.86 priority-based scheduling: Scheduling in which the selection of a running thread is determined by the priority of the runnable threads.

2.2.2.87 privilege: See appropriate privileges in 2.2.2.4.

2.2.2.88 process: An address space with one or more threads executing within that address space, and the required system resources for those threads.

A process is created by another process issuing the *fork*() function. The process that issues *fork*() is known as the parent process, and the new process created by the *fork*() is known as the child process.

Many of the system resources defined by this part of ISO/IEC 9945 are shared among all of the threads within a process. These include the process ID; the parent process ID; the process group ID; the session membership; the real, effective and saved-set user ID; the real, effective and saved-set group ID; the supplementary group IDs; the current working directory; the root directory; the file mode creation mask; and file descriptors.

2.2.2.89 process group: A collection of processes that permits the signaling of related processes.

Each process in the system is a member of a process group that is identified by a process group ID. A newly created process joins the process group of its creator.

2.2.2.90 process group ID: The unique identifier representing a process group during its lifetime.

A process group ID is a positive integer that can be contained in a *pid_t*. It shall not be reused by the system until the process group lifetime ends.

2.2.2.91 process group leader: A process whose process ID is the same as its process group ID.

2.2.2.92 process group lifetime: A period of time that begins when a process group is created and ends when the last remaining process in the group leaves the group, due either to the end of the last process's process lifetime or to the last remaining process calling the *setsid*() or *setpgid*() functions.

2.2.2.93 process ID: The unique identifier representing a process.

A process ID is a positive integer that can be contained in a *pid_t*. A process ID shall not be reused by the system until the process lifetime ends. In addition, if there exists a process group whose process group ID is equal to that process ID, the process ID shall not be reused by the system until the process group lifetime ends. A process that is not a system process shall not have a process ID of 1.

2.2.2.94 process lifetime: The period of time that begins when a process is created and ends when its process ID is returned to the system.

After a process is created with a *fork*() function, it is considered active. At least one thread of control and the address space exist until it terminates. It then enters an inactive state where certain resources may be returned to the system, although some resources, such as the process ID, are still in use. When another process executes a *wait*() or *waitpid*() function for an inactive process, the remaining resources are returned to the system. The last resource to be returned to the system is the process ID. At this time, the lifetime of the process ends.

2.2.2.95 read-only file system: A file system that has implementation-defined characteristics restricting modifications.

2.2.2.96 real group ID: The attribute of a process that, at the time of process creation, identifies the group of the user who created the process.

See group ID in 2.2.2.55. This value is subject to change during the process lifetime, as described in 4.2.2.

2.2.2.97 real user ID: The attribute of a process that, at the time of process creation, identifies the user who created the process.

See user ID in 2.2.2.143. This value is subject to change during the process lifetime, as described in 4.2.2.

2.2.2.98 reentrant function: A function whose effect, when called by two or more threads, is guaranteed to be as if the threads each executed the function one after another in an undefined order, even if the actual execution is interleaved.

2.2.2.99 referenced shared memory object: A shared memory object that is open or has one or more mappings defined on it.

2.2.2.100 region:

1) As relates to the address space of a process, a sequence of addresses.
2) As relates to a file, a sequence of offsets.

2.2.2.101 regular file: A file that is a randomly accessible sequence of bytes, with no further structure imposed by the system.

2.2.2.102 relative pathname: See pathname resolution in 2.3.6.

2.2.2.103 (time) resolution: The minimum time interval that a clock can measure or whose passage a timer can detect.

2.2.2.104 root directory: A directory, associated with a process, that is used in pathname resolution for pathnames that begin with a slash.

2.2.2.105 runnable thread: A thread that is capable of being a running thread, but for which no processor is available.

2.2.2.106 running thread: A thread currently executing on a processor.

There may be more than one such thread in a system at a time in a system with multiple processors.

2.2.2.107 saved set-group-ID: An attribute of a process that allows some flexibility in the assignment of the effective group ID attribute, when the saved set-user-ID option is implemented, as described in 3.1.2 and 4.2.2.

2.2.2.108 saved set-user-ID: An attribute of a process that allows some flexibility in the assignment of the effective user ID attribute, when the saved set-user-ID option is implemented, as described in 3.1.2 and 4.2.2.

2.2.2.109 scheduling: The application of a policy to select a runnable thread to become a running thread, or to alter one or more of the thread lists.

2.2.2.110 scheduling allocation domain: The set of processors on which an individual thread can be scheduled at any given time.

2.2.2.111 scheduling contention scope: A property of a thread that defines the set of threads against which that thread competes for resources.

For example, in a scheduling decision, threads sharing scheduling contention scope compete for processor resources. In this standard, a thread has a scheduling contention scope of either PTHREAD_SCOPE_SYSTEM or PTHREAD_SCOPE_PROCESS.

2.2.2.112 scheduling policy: A set of rules that is used to determine the order of execution of threads to achieve some goal.

In the context of this standard, a scheduling policy affects thread ordering

1) When a thread is a running thread and it becomes a blocked thread
2) When a thread is a running thread and it becomes a preempted thread
3) When a thread is a blocked thread and it becomes a runnable thread

4) When a running thread calls a function that can change the priority or scheduling policy of a thread
5) In other scheduling-policy-defined circumstances

Conforming implementations shall define the manner in which each of the scheduling policies may modify the priorities or otherwise affect the ordering of threads at each of the occurrences listed above. Additionally, conforming implementations shall define at what other circumstances and in what manner each scheduling policy may modify the priorities or affect the ordering of threads.

2.2.2.113 seconds since the Epoch: A value to be interpreted as the number of seconds between a specified time and the Epoch.

A Coordinated Universal Time name (specified in terms of seconds (*tm_sec*), minutes (*tm_min*), hours (*tm_hour*), days since January 1 of the year (*tm_yday*), and calendar year minus 1900 (*tm_year*) is related to a time represented as seconds since the Epoch, according to the expression below.

If the year < 1970 or the value is negative, the relationship is undefined. If the year ≥ 1970 and the value is nonnegative, the value is related to a Coordinated Universal Time name according to the expression:

$$tm\_sec + tm\_min*60 + tm\_hour*3600 + tm\_yday*86\,400 +$$
$$(tm\_year-70)*31\,536\,000 + ((tm\_year-69)/4)*86\,400$$

2.2.2.114 semaphore: A shareable resource that has a nonnegative integral value.

When the value is zero, there is a (possibly empty) set of threads awaiting the availability of the semaphore.

2.2.2.115 semaphore lock operation: An operation that is applied to a semaphore.

If, prior to the operation, the value of the semaphore is zero, the semaphore lock operation shall cause the calling thread to be blocked and added to the set of threads awaiting the semaphore. Otherwise, the value is decremented. See 2.2.2.116.

2.2.2.116 semaphore unlock operation: An operation that is applied to a semaphore.

If, prior to the operation, there are any threads in the set of threads awaiting the semaphore, then some thread from that set shall be removed from the set and become unblocked. Otherwise, the semaphore value is incremented. See 2.2.2.115.

2.2.2.117 session: A collection of process groups established for job control purposes.

Each process group is a member of a session. A process is considered to be a member of the session of which its process group is a member. A newly created process joins the session of its creator. A process can alter its session membership (see 4.3.2). Implementations that support the *setpgid*() function (see 4.3.3) can have multiple process groups in the same session.

2.2.2.118 session leader: A process that has created a session (see 4.3.2).

2.2.2.119 session lifetime: The period between when a session is created and the end of the lifetime of all the process groups that remain as members of the session.

2.2.2.120 shared memory object: An object that represents memory that can be mapped concurrently into the address space of more than one process.

2.2.2.121 signal: A mechanism by which a process may be notified of, or affected by, an event occurring in the system.

Examples of such events include hardware exceptions and specific actions by processes or threads. The term *signal* is also used to refer to the event itself.

2.2.2.122 slash: The literal character "/".

This character is also known as *solidus* in ISO 8859-1 {B34} .

2.2.2.123 supplementary group ID: An attribute of a process used in determining file access permissions.

A process has up to {NGROUPS_MAX} supplementary group IDs in addition to the effective group ID. The supplementary group IDs of a process are set to the supplementary group IDs of the parent process when the process is created. Whether a process's effective group ID is included in or omitted from its list of supplementary group IDs is unspecified.

2.2.2.124 successfully transferred: For a write operation to a regular file, when the system ensures that all data written is readable on any subsequent open of the file (even one that follows a system or power failure) in the absence of a failure of the physical storage medium.

For a read operation, when an image of the data on the physical storage medium is available to the requesting process.

2.2.2.125 synchronized I/O completion: The state of an I/O operation that has either been successfully transferred or diagnosed as unsuccessful.

2.2.2.126 synchronized I/O data integrity completion:

1) For read, when the operation has been completed or diagnosed if unsuccessful. The read is complete only when an image of the data has been successfully transferred to the requesting process. If there were any pending write requests affecting the data to be read at the time that the synchronized read operation was requested, these write requests shall be successfully transferred prior to reading the data.
2) For write, when the operation has been completed or diagnosed if unsuccessful. The write is complete only when the data specified in the write request is successfully transferred and all file system information required to retrieve the data is successfully transferred.

File attributes that are not necessary for data retrieval (access time, modification time, status change time) need not be successfully transferred prior to returning to the calling process.

2.2.2.127 synchronized I/O file integrity completion: Identical to a synchronized I/O data integrity completion with the addition that all file attributes relative to the I/O operation (including access time, modification time, status change time) shall be successfully transferred prior to returning to the calling process.

2.2.2.128 synchronized I/O operation: An I/O operation performed on a file that provides the application assurance of the integrity of its data and files.

2.2.2.129 synchronous I/O operation: An I/O operation that causes the process requesting the I/O to be blocked from further use of the processor until that I/O operation completes.

NOTE — A synchronous I/O operation does not perforce imply synchronized I/O data integrity completion or synchronized I/O file integrity completion.

2.2.2.130 synchronously generated signal: A signal that is attributable to a specific thread.

For example, a thread executing an illegal instruction or touching invalid memory causes a synchronously generated signal. Being synchronous is a property of how the signal was generated and not a property of the signal number.

2.2.2.131 system: An implementation of this part of ISO/IEC 9945 .

2.2.2.132 system crash: An interval initiated by an unspecified circumstance that causes all processes (possibly other than special system processes) to be terminated in an undefined manner, after which any changes to the state and contents of files created or written to by a Conforming POSIX.1 Application prior to the interval are undefined, except as required elsewhere in this standard.

2.2.2.133 system process: An object, other than a process executing an application, that is defined by the system and has a process ID.

2.2.2.134 system reboot: An implementation defined sequence of events that may result in the loss of transitory data, i.e., data that is not saved in permanent storage.

This includes message queues, shared memory, semaphores, and processes.

2.2.2.135 terminal [terminal device]: A character special file that obeys the specifications of 7.1.

2.2.2.136 thread: A single flow of control within a process.

Each thread has its own thread ID, scheduling priority and policy, *errno* value, thread-specific key/value bindings, and the required system resources to support a flow of control. Anything whose address may be determined by a thread, including but not limited to static variables, storage obtained via *malloc*(), directly addressable storage obtained through implementation-supplied functions, and automatic variables shall be accessible to all threads in the same process.

2.2.2.137 thread ID: A unique value of type *pthread_t* that identifies each thread during its lifetime in a process.

2.2.2.138 thread list: An ordered set of runnable threads that all have the same ordinal value for their priority.

The ordering of threads on the list is determined by a scheduling policy or policies. The set of thread lists includes all runnable threads in the system.

2.2.2.139 thread-safe: A function that may be safely invoked concurrently by multiple threads.

Each function defined by this standard is thread-safe unless explicitly stated otherwise. An example is any "pure" function (a function that holds a mutex locked while it is accessing static storage or objects shared among threads).

2.2.2.140 thread-specific data key: A process global handle of type *pthread_key_t* that is used for naming thread-specific data.

Although the same key value may be used by different threads, the values bound to the key by *pthread_setspecific*() and accessed by *pthread_getspecific*() are maintained on a per-thread basis and persist for the life of the calling thread.

2.2.2.141 timer: An object that can notify a process when the time as measured by a particular clock has reached or passed a specified value, or when a specified amount of time, as measured by a particular clock, has passed.

2.2.2.142 timer overrun: A condition that occurs each time a timer, for which there is already an expiration signal queued to the process, expires.

2.2.2.143 user ID: A nonnegative integer, which can be contained in an object of type *uid_t*, that is used to identify a system user.

When the identity of a user is associated with a process, a user ID value is referred to as a real user ID, an effective user ID, or an (optional) saved set-user-ID.

2.2.2.144 user name: A string that is used to identify a user, as described in 9

2.2.2.145 working directory [current working directory]: A directory, associated with a process, that is used in pathname resolution for pathnames that do not begin with a slash.

2.2.3 Abbreviations

For the purposes of this part of ISO/IEC 9945, the following abbreviations apply:

2.2.3.1 C Standard: ISO/IEC 9899, *Programming languages—C* {2}.

2.2.3.2 IRV: The International Reference Version coded character set described in ISO/IEC 646 {1}.

2.2.3.3 POSIX.1: This part of ISO/IEC 9945.

2.3 General Concepts

NOTE — As new concepts are added to this clause, they are appended rather than inserted in English alphabetical order. This ensures that external uses of the subclause numbers (such as in standards conformance documents and test method standards) are not invalidated by extensions to this standard.

2.3.1 extended security controls: The access control (see *file access permissions*) and privilege (see *appropriate privileges* in 2.2.2.4) mechanisms have been defined to allow implementation-defined extended security controls. These permit an implementation to provide security mechanisms to implement different security policies than described in this part of ISO/IEC 9945. These mechanisms shall not alter or override the defined semantics of any of the functions in this part of ISO/IEC 9945.

2.3.2 file access permissions: The standard file access control mechanism uses the file permission bits, as described below. These bits are set at file creation by *open*(), *creat*(), *mkdir*() and *mkfifo*() and are changed by *chmod*(). These bits are read by *stat*() or *fstat*().

Implementations may provide *additional* or *alternate* file access control mechanisms, or both. An additional access control mechanism shall only further restrict the access permissions defined by the file permission bits. An alternate access control mechanism shall:

1) Specify file permission bits for the file owner class, file group class, and file other class of the file, corresponding to the access permissions, to be returned by *stat*() or *fstat*().
2) Be enabled only by explicit user action, on a per-file basis by the file owner or a user with the appropriate privilege.
3) Be disabled for a file after the file permission bits are changed for that file with *chmod*(). The disabling of the alternate mechanism need not disable any additional mechanisms defined by an implementation.

Whenever a process requests file access permission for read, write, or execute/search, if no additional mechanism denies access, access is determined as follows:

1) If a process has the appropriate privilege:
   a) If read, write, or directory search permission is requested, access is granted.
   b) If execute permission is requested, access is granted if execute permission is granted to at least one user by the file permission bits or by an alternate access control mechanism; otherwise, access is denied.
2) Otherwise:
   a) The file permission bits of a file contain read, write, and execute/search permissions for the file owner class, file group class, and file other class.
   b) Access is granted if an alternate access control mechanism is not enabled and the requested access permission bit is set for the class (file owner class, file group class, or file other class) to which the process belongs, or if an alternate access control mechanism is enabled and it allows the requested access; otherwise, access is denied.

2.3.3 file hierarchy: Files in the system are organized in a hierarchical structure in which all of the nonterminal nodes are directories and all of the terminal nodes are any other type of file. Because multiple directory entries may refer to the same file, the hierarchy is properly described as a “directed graph.”

2.3.4 filename portability: Filenames should be constructed from the portable filename character set because the use of other characters can be confusing or ambiguous in certain contexts.

2.3.5 file times update: Each file has three distinct associated time values: *st_atime*, *st_mtime*, and *st_ctime*. The *st_atime* field is associated with the times that the file data is accessed; *st_mtime* is associated with the times that the file data is modified; and *st_ctime* is associated with the times that file status is changed. These values are returned in the file characteristics structure, as described in 5.6.1.

Any function in this part of ISO/IEC 9945 that is required to read or write file data or change the file status indicates which of the appropriate time-related fields are to be “marked for update.” If an implementation of such a function marks for update a time-related field not specified by this part of ISO/IEC 9945, this shall be documented, except that any changes caused by pathname resolution need not be documented. For the other functions in this part of ISO/IEC 9945 (those that are not explicitly required to read or write file data or change file status, but that in some implementations happen to do so), the effect is unspecified.

An implementation may update fields that are marked for update immediately, or it may update such fields periodically. When the fields are updated, they are set to the current time and the update marks are cleared. All fields that are marked for update shall be updated when the file is no longer open by any process, or when a *stat*() or *fstat*() is performed on the file. Other times at which updates are done are unspecified. Updates are not done for files on read-only file systems.

2.3.6 pathname resolution: Pathname resolution is performed for a process to resolve a pathname to a particular file in a file hierarchy. There may be multiple pathnames that resolve to the same file.

Each filename in the pathname is located in the directory specified by its predecessor (for example, in the pathname fragment “a/b”, file “b” is located in directory “a”). Pathname resolution fails if this cannot be accomplished. If the pathname begins with a slash, the predecessor of the first filename in the pathname is taken to be the root directory of the process (such pathnames are referred to as absolute pathnames). If the pathname does not begin with a slash, the predecessor of the first filename of the pathname is taken to be the current working directory of the process (such pathnames are referred to as "relative pathnames").

The interpretation of a pathname component is dependent on the values of {NAME_MAX} and {_POSIX_NO_TRUNC} associated with the path prefix of that component. If any pathname component is longer than {NAME_MAX}, and {_POSIX_NO_TRUNC} is in effect for the path prefix of that component (see 5.7.1), the implementation shall consider this an error condition. Otherwise, the implementation shall use the first {NAME_MAX} bytes of the pathname component.

The special filename, dot, refers to the directory specified by its predecessor. The special filename, dot-dot, refers to the parent directory of its predecessor directory. As a special case, in the root directory, dot-dot may refer to the root directory itself.

A pathname consisting of a single slash resolves to the root directory of the process. A null pathname is invalid.

2.3.7 concurrent execution: Functions that suspend the execution of the calling thread shall not cause the execution of other threads to be indefinitely suspended.

2.3.8 memory synchronization: Applications shall ensure that access to any memory location by more than one thread of control (threads or processes) is restricted such that no thread of control can read or modify a memory location while another thread of control may be modifying it. Such access is restricted using functions that synchronize thread execution and also synchronize memory with respect to other threads. The following functions synchronize memory with respect to other threads:

fork()	*pthread_mutex_unlock*()	*sem_post*()
pthread_create()	*pthread_cond_wait*()	*sem_trywait*()
pthread_join()	*pthread_cond_timedwait*()	*sem_wait*()
pthread_mutex_lock()	*pthread_cond_signal*()	*wait*()
pthread_mutex_trylock()	*pthread_cond_broadcast*()	*waitpid*()

Unless explicitly stated otherwise, if one of the above functions returns an error, it is unspecified whether the invocation causes memory to be synchronized.

Applications may allow more than one thread of control to read a memory location simultaneously.

2.3.9 thread-safety: All functions defined by POSIX.1 and the C Standard {2} shall be thread-safe, except that the following functions need not be thread-safe: *asctime*(), *ctime*(), *getc_unlocked*(), *getchar_unlocked*(), *getgrgid*(), *getgrnam*(), *getlogin*(), *getpwnam*(), *getpwuid*(), *gmtime*(), *localtime*(), *putc_unlocked*(), *putchar_unlocked*(), *rand*(), *readdir*(), *strtok*(), and *ttyname*(). The functions *ctermid*() and *tmpnam*() need not be thread-safe if a **NULL** argument is passed to the function. Implementations shall provide internal synchronization as necessary in order to satisfy this requirement.

2.4 Error Numbers

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

Some functions may provide the error number in a variable accessed through the symbol *errno*. The symbol *errno* is defined by including the header `<errno.h>`, as specified by the C Standard {2}.

The value of *errno* should only be examined when it is indicated to be valid by the return value of a function. No function defined in this standard shall set *errno* to zero to indicate an error. For each thread of a process, the value of *errno* shall not be affected by function calls or assignments to *errno* by other threads.

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

If more than one error occurs in processing a function call, this part of ISO/IEC 9945 does not define in what order the errors are detected; therefore, any one of the possible errors may be returned.

Implementations may support additional errors not included in this clause, may generate errors included in this clause under circumstances other than those described in this clause, or may contain extensions or limitations that prevent some errors from occurring. The Errors subclause in each function description specifies which error conditions shall be detected by all implementations and which may be optionally detected by an implementation. Each implementation shall document, in the conformance document, situations in which each of the optional conditions are detected. If no error condition is detected, the action requested shall be successful. Implementations may contain extensions or limitations that prevent some specified errors from occurring.

Implementations may generate error numbers listed in this clause under circumstances other than those described, if and only if all those error conditions can always be treated identically to the error conditions as described in this part of ISO/IEC 9945. Implementations may support additional errors not listed in this clause, but shall not generate a different error number from one required by this part of ISO/IEC 9945 for an error condition described in this part of ISO/IEC 9945. For interfaces under the {POSIX_THREADS} option for which [EINTR] is not listed as a possible error return in this; standard, an implementation shall not return an error code of [EINTR].

The following symbolic names identify the possible error numbers, in the context of functions specifically defined in this part of ISO/IEC 9945; these general descriptions are more precisely defined in the Errors subclauses of functions that return them. Only these symbolic names should be used in programs, since the actual value of an error number is unspecified. All values listed in this clause shall be unique. The values for these names shall be found in the header `<errno.h>`. The actual values are unspecified by this part of ISO/IEC 9945.

[E2BIG]	Arg list too long	The sum of the number of bytes used by the new process image's argument list and environment list was greater than the system-imposed limit of {ARG_MAX} bytes.
[EACCES]	Permission denied	An attempt was made to access a file in a way forbidden by its file access permissions.
[EAGAIN]	Resource temporarily unavailable	This is a temporary condition, and later calls to the same routine may complete normally.
[EBADF]	Bad file descriptor	A file descriptor argument was out of range,referred to no open file, or a read (write) request was made to a file that was only open for writing (reading).
[EBADMSG]	Bad message	The implementation has detected a corrupted message.
[EBUSY]	Resource busy	An attempt was made to use a system resource that was not available at the time because it was being used by a process in a manner that would have conflicted with the request being made by this process.
[ECANCELED]	Operation canceled	The associated asynchronous operation was canceled before completion.

[ECHILD] No child processes

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

[EDEADLK]

Resource deadlock avoided

An attempt was made to lock a system resource that would have resulted in a deadlock situation.

[EDOM] Domain error

Defined in the C Standard {2}; an input argument was outside the defined domain of the mathematical function.

[EEXIST] File exists

An existing file was specified in an inappropriate context; for instance, as the new link name in a *link*() function.

[EFAULT] Bad address

The system detected an invalid address in attempting to use an argument of a call. The reliable detection of this error is implementation defined; however, implementations that do detect this condition shall use this value.

[EFBIG] File too large

The size of a file would exceed an implementation-define maximum file size.

[EINPROGRESS]

Operation in progress

An asynchronous operation has not yet completed.

[EINTR] Interrupted function call

An asynchronous signal (such as SIGINT or SIGQUIT; see the description of header `<signal.h>` in 3.3.1.1) 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 error condition.

[EINVAL] Invalid argument

Some invalid argument was supplied. [For example, specifying an undefined signal to a *signal*() or *kill*() function].

[EIO] Input/output error

Some physical input or output error 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.

[EISDIR] Is a directory

An attempt was made to open a directory with write mode specified.

[EMFILE] Too many open files

An attempt was made to open more than the maximum number of {OPEN_MAX} file descriptors allowed in this process.

[EMLINK] Too many links

An attempt was made to have the link count of a single file exceed {LINK_MAX}.

[EMSGSIZE]

Inappropriate message buffer length

[ENAMETOOLONG]

Filename too long

The size of a pathname string exceeded {PATH_MAX}, or a pathname component was longer than {NAME_MAX} and {_POSIX_NO_TRUNC} was in effect for that file.

[ENFILE] Too many open files in system

Too many files are currently open in the system. The system reached its predefined limit for simultaneously open files and temporarily could not accept requests to open another one.

[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 did not exist, or the pathname was an empty string.

[ENOEXEC]

Exec format error

A request was made to execute a file that, although it had the appropriate permissions, was 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 was reached, and no more were available at that time.

[ENOMEM]

Not enough space

The new process image required more memory than was allowed by the hardware or by system-imposed memory management constraints.

[ENOSPC] No space left on device

During a *write*() function on a regular file, or when extending a directory, there was no free space left on the device.

[ENOSYS] Function not implemented

An attempt was made to use a function that is not available in this implementation.

[ENOTDIR]

Not a directory

A component of the specified pathname existed, but it was not a directory, when a directory was expected.

[ENOTEMPTY]

Directory not empty

A directory with entries other than dot and dot-dot was supplied when an empty directory was expected.

[ENOTSUP]	
	Not supported
	The implementation does not support this feature of the standard.
[ENOTTY]	Inappropriate I/O control operation
	A control function was attempted for a file or a special file for which the operation was inappropriate.
[ENXIO]	No such device or address
	Input or output on a special file referred to a device that did not exist, or made a request beyond the limits of the device. This error may also occur when, for example, a tape drive is not online or a disk pack is not loaded on a drive.
[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 pipe or FIFO for which there was no process to read the data.
[ERANGE]	Result too large
	Defined in the C Standard {2}; the result of the function was too large to fit in the available space.
[EROFS]	Read-only file system
	An attempt was made to modify a file or directory on a file system that was read-only at that time.
[ESPIPE]	Invalid seek
	An *lseek*() function was issued on a pipe or FIFO.
[ESRCH]	No such process
	No process could be found corresponding to that specified by the given process ID.
[ETIMEDOUT]	
	Operation timed out
	The time limit associated with the operation was exceeded before the operation completed.
[EXDEV]	Improper link
	A link to a file on another file system was attempted.

2.5 Primitive System Data Types

Some data types used by the various system functions are not defined as part of this part of ISO/IEC 9945, but are defined by the implementation. These types are then defined in the header `<sys/types.h>`, which contains definitions for at least the types shown in Table 2.1.

All of the types listed in Table 2.1 shall be arithmetic types; *pid_t*, *ssize_t*, and *off_t* shall be signed arithmetic types. The type *ssize_t* shall be capable of storing values in the range from –1 to {SSIZE_MAX}, inclusive. The types *size_t* and *ssize_t* shall also be defined in the header `<unistd.h>`.

Additional unspecified type symbols ending in *_t* may be defined in any header specified by POSIX.1. The visibility of such symbols need not be controlled by any feature test macro other than _POSIX_C_SOURCE.

There are no defined comparison or assignment operators for the types *pthread_attr_t*, *pthread_cond_t*, *pthread_condattr_t*, *pthread_mutex_t*, and *pthread_mutexattr_t*.

Table 2.1—Primitive System Data Types

Defined Type	**Description**
dev_t	Used for device numbers.
gid_t	Used for group IDs.
ino_t	Used for file serial numbers.
mode_t	Used for some file attributes, for example file type, file access permissions.
nlink_t	Used for link counts.
off_t	Used for file sizes.
pid_t	Used for process IDs and process group IDs.
pthread_t	Used to identify a thread.
pthread_attr_t	Used to identify a thread attributes object.
pthread_mutex_t	Used for mutexes.
pthread_mutexattr_t	Used to identify a mutex attributes object.
pthread_cond_t	Used for condition variables.
pthread_condattr_t	Used to identify a condition attributes object.
pthread_key_t	Used for thread-specific data keys.
pthread_once_t	Used for dynamic package initialization.
size_t	As defined in the C Standard {2}.
ssize_t	Used by functions that return a count of bytes (memory space) or an error indication.
uid_t	Used for user IDs.

2.6 Environment Description

An array of strings called the *environment* is made available when a process begins. This array is pointed to by the external variable *environ*, which is defined as:

```
extern char **environ;
```

These strings have the form “*name*=*value*”; *names* shall not contain the character '='. There is no meaning associated with the order of the strings in the environment. If more than one string in a process's environment has the same *name*, the consequences are undefined. The following names may be defined and have the indicated meaning if they are defined:

HOME	The name of the initial working directory of the user from the user database (see the description of the header `<pwd.h>` in 9.2.2.1).
LANG	The name of the locale to use for locale categories when both **LC_ALL** and the corresponding environment variable ('beginning with “**LC_**”) do not specify a locale.

LC_ALL	The name of the locale to be used to override any values for locale categories specified by the setting of **LANG** or any environment variables beginning with "**LC_**".
LC_COLLATE	The name of the locale for collation information.
LC_CTYPE	The name of the locale for character classification.
LC_MONETARY	The name of the locale containing monetary-related numeric editing information.
LC_NUMERIC	The name of the locale containing numeric editing (i.e., radix character) information.
LC_TIME	The name of the locale for date/time formatting information.
LOGNAME	The login name associated with the current process. The value shall be composed of characters from the portable filename character set. NOTE — An application that requires, or an installation that actually uses, characters outside the portable filename character set would not strictly conform to this part of ISO/IEC 9945 . However, it is reasonable to expect that such characters would be used in many countries (recognizing the reduced level of interchange implied by this), and applications or installations should permit such usage where possible. No error is defined by this part of ISO/IEC 9945 for violation of this condition.
PATH	The sequence of path prefixes that certain functions apply in searching for an executable file known only by a filename (a pathname that does not contain a slash). The prefixes are separated by a colon (:). When a nonzero-length prefix is applied to this filename, a slash is inserted between the prefix and the filename. A zero-length prefix is a special prefix that indicates the current working directory. It appears as two adjacent colons (::), as an initial colon preceding the rest of the list, or as a trailing colon following the rest of the list. The list is searched from beginning to end until an executable program by the specified name is found. If the pathname being sought contains a slash, the search through the path prefixes is not performed.
TERM	The terminal type for which output is to be prepared. This information is used by commands and application programs wishing to exploit special capabilities specific to a terminal.
TZ	Time zone information. The format of this string is defined in 8.1.1.

Environment variable *names* used or created by an application should consist solely of characters from the portable filename character set. Other characters may be permitted by an implementation; applications shall tolerate the presence of such names. Upper- and lowercase letters retain their unique identities and are not folded together. System-defined environment variable names should begin with a capital letter or underscore and be composed of only capital letters, underscores, and numbers.

The *values* that the environment variables may be assigned are not restricted except that they are considered to end with a null byte, and the total space used to store the environment and the arguments to the process is limited to {ARG_MAX} bytes.

Other *name*=*value* pairs may be placed in the environment by manipulating the *environ* variable subject to the restrictions specified in 3.1.2 or by using *envp* arguments when creating a process (see 3.1.2).

2.7 C Language Definitions

2.7.1 Symbols From the C Standard

The following terms and symbols used in this part of ISO/IEC 9945 are defined in the C Standard {2}: *NULL, byte, array of char, clock_t, header, null character, string, time_t*. The type *clock_t* shall be capable of representing all integer values from zero to the number of clock ticks in 24 h.

The term **NULL** *pointer* in this part of ISO/IEC 9945 is equivalent to the term *null pointer* used in the C Standard {2}. The symbol **NULL** shall be declared in `<unistd.h>` with the same value as required by the C Standard {2}, in addition to several headers already required by the C Standard {2}.

Additionally, the reservation of symbols that begin with an underscore applies:

1) All external identifiers that begin with an underscore are reserved.
2) All other identifiers that begin with an underscore and either an uppercase letter or another underscore are reserved.
3) If the program defines an external identifier with the same name as a reserved external identifier, even in a semantically equivalent form, the behavior is undefined.

Certain other namespaces are reserved by the C Standard {2}. These reservations apply to this part of ISO/IEC 9945 as well. Additionally, the C Standard {2} requires that it be possible to include a header more than once and that a symbol may be defined in more than one header. This requirement is also made of headers for this part of ISO/IEC 9945.

2.7.2 POSIX.1 Symbols

Certain symbols in this part of ISO/IEC 9945 are defined in headers. Some of those headers could also define other symbols than those defined by this part of ISO/IEC 9945, potentially conflicting with symbols used by the application. Also, this part of ISO/IEC 9945 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 this part of ISO/IEC 9945, and other standards may define additional feature test macros. Feature test macros shall be defined in the compilation of an application before an `#include` of any header where a symbol should be visible to some, but not all, applications. If the definition of the macro does not precede the #include, the result is undefined.

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

Implementations may add symbols to the headers shown in Table 2.2, provided the identifiers for those symbols begin with the corresponding reserved prefixes in Table 2.2. Similarly, implementations may add symbols to the headers in Table 2.2 that end in the string indicated as a reserved suffix as long as the reserved suffix is in that part of the name considered significant by the implementation. This shall be in addition to any reservations made in the C Standard {2}.

If any header defined by this part of ISO/IEC 9945 is included, all symbols with the suffix_t are reserved for use by the implementation, both before and after the `#include` directive.

After the last inclusion of a given header, an application may use any of the symbol classes reserved in Table 2.2 for its own purposes, as long as the requirements in the note to Table 2.2 are satisfied, noting that the symbol declared in the header may become inaccessible.

Future revisions of this part of ISO/IEC 9945, and other POSIX standards, are likely to use symbols in these same reserved spaces.

Table 2.2—Reserved Header Symbols

Header	Key	Reserved Prefix	Reserved Suffix
`<aio.h>`	1	`aio_`	
	1	`lio_`	
	2	`AIO_`	
	2	`LIO_`	
`<dirent.h>`	1	`d_`	
`<fcntl.h>`	1	`l_`	
	2	`F_`	
	2	`O_`	
	2	`S_`	
`<grp.h>`	1	`gr_`	
`<limits.h>`	1		`_MAX`
`<locale.h>`	2	`LC_[A–Z]`	
`<mqueue.h>`	1	`mq_`	
	2	`MQ_`	
`<pthread.h>`	1	`pthread_`	
	2	`PTHREAD_`	
`<pwd.h>`	1	`pw_`	
`<sched.h>`	1	`sched_`	
	2	`SCHED_`	
`<semaphore.h>`	1	`sem_`	
	2	`SEM_`	
`<signal.h>`	1	`sa_`	
	2	`SIG_`	
	2	`SA_`	
	1	`si_`	
	2	`SI_`	
	1	`sigev_`	
	2	`SIGEV_`	
	1	`sival_`	

Table 2.2—Reserved Header Symbols (Continued)

Header	Key	Reserved Prefix	Reserved Suffix
`<sys/mman.h>`	1	`shm_`	
	2	`MAP_`	
	2	`MCL_`	
	2	`MS_`	
	2	`PROT_`	
`<sys/stat.h>`	1	`st_`	
	2	`S_`	
`<sys/times.h>`	1	`tms_`	
`<termios.h>`	1	`c_`	
	2	`V`	
	2	`I`	
	2	`O`	
	2	`TC`	
	2	`B [0–9]`	
`<time.h>`	1	`clock_`	
	1	`timer_`	
	1	`it_`	
	1	`tv_`	
	2	`CLOCK_`	
	2	`TIMER_`	
any POSIX.1 header included	1		`_t`

NOTE — The notation "[0–9]" indicates any digit and "[A–Z]" any uppercase character in the portable filename character set. The Key values, are:

1) Prefixes and suffixes of symbols that shall not be declared or #defined by the application.
2) Prefixes and suffixes of symbols that shall be preceded in the application with a `#undef` of that symbol before any other use.

In addition, implementations may 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 header `<fcntl.h>` may contain the following symbols in addition to those specifically required elsewhere in POSIX.1:

SEEK_CUR	S_IRUSR	S_ISCHR	S_ISREG	S_IWUSR
SEEK_END	S_IRWXG	S_ISDIR	S_ISUID	S_IXGRP
SEEK_SET	S_IRWXO	S_ISFIFO	S_IWGRP	S_IXOTH

S_IRGRP	S_IRWXU	S_ISGID	S_IWOTH	S_IXUSR
S_IROTH	S_ISBLK			

In addition, an implementation may define the symbols “cuserid” in `<unistd.h>` and “L_cuserid” in `<stdio.h>`.

The following feature test macro is defined:

Name	Description
_POSIX_C_SOURCE	When an application includes a header described by this part of ISO/IEC 9945, and when this feature test macro is defined to have at least the value 199506L: 1) All symbols required by this part of ISO/IEC 9945 to appear when the header is included shall be made visible. 2) Symbols that are explicitly permitted, but not required, by this part of ISO/IEC 9945 to appear in that header (including those in reserved name spaces) may be made visible. 3) Additional symbols not required or explicitly permitted by this part of ISO/IEC 9945 to be in that header shall not be made visible, except when enabled by another feature test macro or by having defined _POSIX_C_SOURCE with a value larger than 199506L.

The exact meaning of feature test macros depends on the type of C language support chosen: C Standard Language-Dependent Support and Common-Usage-Dependent Support, described in the following two subclauses.

2.7.2.1 C Standard Language-Dependent Support

If there are no feature test macros present in a program, the implementation shall make visible only those identifiers specified as reserved identifiers in the C Standard {2}, permitting the reservations of the symbols and namespace defined in 2.7.1. For each feature test macro present, only the symbols specified by that feature test macro plus those of the C Standard {2} shall be defined when a header is included.

2.7.2.2 Common-Usage-Dependent Support

If the feature test macro _POISIX_C_SOURCE is not defined in a program, the set of symbols defined in each header that are beyond the requirements of this part of ISO/IEC 9945 is unspecified.

If _POSIX_C_SOURCE is defined before any header is included, no symbols other than those from the C Standard {2} and those made visible by feature test macros defined for the program (including _POSIX_C_SOURCE) will be visible. Symbols from the namespace reserved for the implementation, as defined by the C Standard {2}, are also pemitted. The symbols beginning with two underscores are examples of this.

If _POSIX_C_SOURCE is not defined before any header is included, the behavior is undefined.

2.7.3 Headers and Function Prototypes

Implementations claiming C Standard {2} Language-Dependent Support shall declare function prototypes for all functions.

Implementations claiming Common-Usage C Language-Dependent Support shall declare the result type for all funcitons not returning a “plain” *int*.

For functions described in the C Standard {2} and included by refenrence in Section 8 (whether or not they are further described in htis part of ISO/IEC 9945), these prototypes or declarations (if declared) shall appear in the headers defined for then in the C Standard {2}. For other functons in htis part of ISO/IEC 9945, the prototypes or declarations shall appear in the headers listed below. If a functon is defined by this part ISO/IEC 9945, is not declared in the C Standard {2}, and is not listed below, it shall have its prototype or declaration (if required) appear in `<unistd.h>`, which shall be `#include`-ed by the application before using any funciton. The requirements about the visibility of symbols in 2.7.2 shall be honored.

`<aio.h>`	*aio_read*(), *aio_write*(), *aio_error*(),
	aio_return(), *aio_cancel*(), *aio_suspend*, *aio_fsync*()
`<dirent.h>`	*opendir*(), *readdir*(), *readdir_r*(), *rewinddir*(), *closedir*()
`<fcntl.h>`	*open*(), *creat*(), *fcntl*
`<grp.h>`	*getgrgid*(), *getgrnam*()
`<mqueue.h>`	*mq_open*(), *mq_close*(), *mq_unlink*(), *mq_send*()
	mq_recieve(), *mq_notify*(), *mq_getattr*(), *mq_setattr*()
`<pthread.h>`	*pthread_attr_init*(), *pthread_attr_destroy*(),
	pthread_attr_setdetachstate(),
	pthread_attr_setinheritsched(),
	pthread_attr_setschedparam(),
	pthread_attr_setschedpolicy(), *pthread_attr_setscope*(),
	pthread_attr_setstacksize(),
	pthread_attr_setstackaddr(),
	pthread_attr_getdetachstate(),
	pthread_attr_getinheritsched(),
	pthread_attr_getschedparam(),
	pthread_attr_getschedpolicy(), *pthread_attr_getscope*(),
	pthread_attr_getstacksize(),
	pthread_attr_getstackaddr(), *pthread_create*(),
	pthread_detach(), *pthread_equal*(), *pthread_exit*(),
	pthread_join(), *pthread_kill*(), *pthread_sigmask*(),
	pthread_once(), *pthread_self*(),
	pthread_mutexattr_init(), *pthread_mutexattr_destroy*(),
	pthread_mutexattr_getprioceiling(),
	pthread_mutexattr_getprotocol(),
	pthread_mutexattr_setprioceiling(),
	pthread_mutexattr_setprotocol(),
	pthread_mutex_destroy(),
	pthread_mutex_getprioceiling(), *pthread_mutex_init*(),
	pthread_mutex_lock(), *pthread_mutex_setprioceiling*(),

	pthread_mutex_trylock(), *pthread_mutex_unlock*(),
	pthread_condattr_init(), *pthread_condattr_destroy*(),
	pthread_cond_broadcast(), *pthread_cond_destroy*(),
	pthread_cond_init(), *pthread_cond_signal*(),
	pthread_cond_timedwait(), *pthread_cond_wait*(),
	pthread_key_create(), *pthread_key_delete*(),
	pthread_setspecific(), *pthread_getspecific*(),
	pthread_setcancelstate(), *pthread_setcanceltype*(),
	pthread_testcancel(), *pthread_cleanup_pop*(),
	pthread_cleanup_push()
`<pwd.h>`	*getpwuid*(), *getpwnam*()
`<sched.h>`	*sched_setparam*(), *sched_getparam*(),
	sched_setscheduler(), *sched_getscheduler*(),
	sched_yield(), *sched_get_priority_max*(),
	sched_get_priority_min(), *sched_get_rr_interval*()
`<semaphore.h>`	*sem_init*(), *sem_destroy*(), *sem_open*(), *sem_close*(),
	sem_unlink(), *sem_wait*(), *sem_trywait*(), *sero_post*(),
	sem_getvalue()
`<setjmp.h>`	*sigsetjmp*(), *siglongjmp*()
`<signal.h>`	*kill*(), *sigemptyset*(), *sigfillset*(), *sigaddset*(), *sigdelset*(),
	sigismember(), *sigaction*(), *sigprocmask*(), *sigpending*(),
	sigsuspend(), *sigqueue*(), *sigtimedwait*(), *sigwait*()
	sigwaitinfo()
`<stdio.h>`	*ctermid*(), *fileno*(), *fdopen*(), *flockfile*(), *funlockfile*(),
	getc_unlocked(), *getchar_unlocked*(), *putc_unlocked*(),
	putchar_unlocked()
`<sys/mman.h>`	*mlockall*(), *munlockall*(), *mlock*(), *munlock*(), *mmap*(),
	munmap(), *mprotect*(), *msync*(), *shm_open*(),
	shm_unlink()
`<sys/stat.h>`	*umask*(), *mkdir*(), *mkfifo*(), *stat*(), *fstat*(), *chmod*(),
	fchmod()
`<stdlib.h>`	*rand_r*()
`<string.h>`	*strtok_r*()
`<sys/times.h>`	*times*()
`<sys/utsname.h>`	*uname*()
`<sys/wait.h>`	*wait*(), *waitpid*()

`<termios.h>`	*cfgetospeed*(), *cfsetospeed*(), *cfgetispeed*(), *cfsetispeed*(), *tcgetattr*(), *tcsetattr*(), *tcsendbreak*(), *tcdrain*(), *tcflush*(), *tcflow*()
`<time.h>`	*tzset*(), *time*(), *ctime_r*(), *gmtime_r*(), *localtime_r*(), *clock_settime*(), *clock_gettime*(), *clock_getres*(), *timer_create*(), *timer_delete*(), *timer_settime*(), *timer_gettime*(), *timer_getoverrun*(), *nanosleep*()
`<utime.h>`	*utime*()

The declarations in the headers shall follow the proper form for the C language option chosen by the implementation. Additionally, pointer arguments that refer to objects not modified by the function being described are declared with `const` qualifying the type to which it points. Implementations claiming Common-Usage C conformance to this part of ISO/IEC 9945 may ignore the presence of this keyword and need not include it in any function declarations. Implementations claiming conformance using the C Standard {2} shall use the `const` modifier as indicated in the prototypes they provide.

Implementations claiming conformance using Common-Usage C may use equivalent implementation-defined constructs when void is used as a result type for a function prototype. They may also use *int* when a function result is declared *ssize_t*.

Neither the names of the formal parameters nor their types, as they appear in an implementation, are specified by this part of ISO/IEC 9945. The names are used within this part of ISO/IEC 9945 as a notational mechanism. However, any declaration provided by an implementation shall accept all actual parameter types that a declaration lexically identical to one in this part of ISO/IEC 9945 shall accept, including the effects of both type conversion and checking for the number of arguments implied by the presence of a filled-out prototype. The implementation's declaration shall not cause a syntax error if an application provides a prototype lexically identical to one in this part of ISO/IEC 9945. It is not a requirement that nonconforming parameters to functions that may be used by an application be diagnosed by an implementation, except as specifically required by this part of ISO/IEC 9945 or the C Standard {2}, as applicable. Where the C Standard {2} has a more restrictive requirement for a function defined by that standard, that requirement shall be honored, and this exception does not apply.

2.8 Numerical Limits

The following subclauses list magnitude limitations imposed by a specific implementation. The braces notation, {LIMIT}, is used in this part of ISO/IEC 9945 to indicate these values, but the braces are n* part of the name.

2.8.1 C Language Limits

The following limits used in this part of ISO/IEC 9945 are defined in the C Standard {2}: {CHAR_BIT}, {CHAR_MAX}, {CHAR_MIN}, {INT_MAX}, {INT_MIN}, {LONG_MAX}, {LONG_MIN}, {MB_LEN_MAX}, {SCHAR_MAX}, {SCHAR_MIN}, {SHRT_MAX}, {SHRT_MIN}, {UCHAR_MAX}, {UINT_MAX}, {ULONG_MAX}, {USHRT_MAX}.

2.8.2 Minimum Values

The symbols in Table 2.3 shall be defined in `<limits.h>` with the values shown. These are symbolic names for the most restrictive value for certain features on a system conforming to this part of ISO/IEC 9945. Related symbols are defined elsewhere in this part of ISO/IEC 9945, which reflect the actual implementation and which need not be as restrictive. A conforming implementation shall provide values at least this large. A portable application shall not require a larger value for correct operation.

Table 2.3—Minimum Values

Name	Description	Value
{_POSIX_AIO_LISTIO_MAX}	The number of I/O operations that can be specified in a list I/O call.	2
{_POSIX_AIO_MAX}	The number of outstanding asynchronous I/O operations.	1
{_POSIX_ARG_MAX}	The length of the arguments for one of the *exec*functions, in bytes, including environment data.	4096
{_POSIX_CHILD_MAX}	The number of simultaneous processes per real user ID.	6
{_POSIX_DELAYTIMER_MAX}	The number of timer expiration overruns.	32
{_POSIX_LINK_MAX}	The value of a file's link count.	8
{_POSIX_LOGIN_NAME_MAX}	The size of the storage required for a login name, in bytes, including the terminating null.	9
{_POSIX_MAX_CANON}	The number of bytes in a terminal canonical input queue.	255
{_POSIX_MAX_INPUT}	The number of bytes for which space will be available in a terminal input queue.	255
{_POSIX_MQ_OPEN_MAX}	The number of message queues that can be open for a single process.	8
{_POSIX_MQ_PRIO_MAX}	The maximum number of message priorities supported by the implementation.	32
{_POSIX_NAME_MAX}	The number of bytes in a filename.	14
{_POSIX_NGROUPS_MAX}	The number of simultaneous supplementary group IDs per process.	0
{_POSIX_OPEN_MAX}	The number of files that one process can have open at one time.	16
{_POSIX_PATH_MAX}	The number of bytes in a pathname.	255
{_POSIX_PIPE_BUF}	The number of bytes that can be written atomically when writing to a pipe.	512
{_POSIX_RTSIG_MAX}	The number of realtime signal numbers reserved for application use.	8
{_POSIX_SEM_NSEMS_MAX}	The number of semaphores that a process may have.	256
{_POSIX_SEM_VALUE_MAX}	The maximum value a semaphore may have.	32 767
{_POSIX_SIGQUEUE_MAX}	The number of queued signals that a process may send and have pending at the receiver(s) at any time.	32
{_POSIX_SSIZE_MAX}	The value that can be stored in an object of type *ssize_t*.	32 767
{_POSIX_STREAM_MAX}	The number of streams that one process can have open at one time.	8
{_POSIX_THREAD_DESTRUCTOR-_ITERATIONS}	The number of attempts made to destroy the thread-specific data values of a thread on thread exit.	4
{_POSIX_THREAD_KEYS_MAX}	The number of data keys per process.	128
{_POSIX_THREAD_THREADS_MAX}	The number of threads per process.	64
{_POSIX_TTY_NAME_MAX}	The size of the storage required for a terminal device name, in bytes, including the terminating null.	9
{_POSIX_TIMER_MAX}	The per-process number of timers.	32
{_POSIX_TZNAME_MAX}	The maximum number of bytes supported for the name of a time zone (not of the **TZ** variable).	3

2.8.3 Run-Time Increasable Values

The magnitude limitations in Table 2.4 shall be fixed by specific implementations.

Table 2.4—Run-Time Increasable Values

Name	Description	Minimum Value
{NGROUPS_MAX}	Maximum number of simultaneous supplementary group IDs per process.	{_POSIX_NGROUPS_MAX}

A Strictly Conforming POSIX. 1 Application shall assume that the value supplied by `<limits.h>` in a specific implementation is the minimum value that pertains whenever the Strictly Conforming POSIX. 1 Application is run under that implementation.[2] A specific instance of a specific implementation may increase the value relative to that supplied by `<limits.h>` for that implementation. The actual value supported by a specific instance shall be provided by the *sysconf*() function.

2.8.4 Run-Time Invariant Values (Possibly Indeterminate)

A definition of one of the values in Table 2.5 shall be omitted from the `<limits.h>` on specific implementations where the corresponding value is equal to or greater than the stated minimum, but is indeterminate.

This might depend on the amount of available memory space on a specific instance of a specific implementation. The actual value supported by a specific instance shall be provided by the *sysconf*() function.

2.8.5 Pathname Variable Values

The values in Table 2.6 may be constants within an implementation or may vary from one pathname to another.

For example, file systems or directories may have different characteristics.

A definition of one of the values from Table 2.6 shall be omitted from `<limits.h>` on specific implementations where the corresponding value is equal to or greater than the stated minimum, but where the value can vary depending on the file to which it is applied. The actual value supported for a specific pathname shall be provided by the *pathconf*() function.

Table 2.5—Run-Time Invariant Values (Possibly Indeterminate)

Name	Description	Minimum Value
{AIO_LISTIO_MAX}	Maximum number of I/O operations in a single list I/O call supported by the implementation.	{_POSIX_AIO_LISTIO_MAX}
{AIO_MAX}	Maximum number of outstanding asynchronous I/O operations supported by the implementation.	{_POSIX_AIO_MAX}
{AIO_PRIO_DELTA_MAX}	The maximum amount by which a process can decrease its asynchronous I/O priority level from its own scheduling priority.	0
{ARG_MAX}	Maximum length of arguments for the *exec* functions, in bytes, including environment data.	{_POSIX_ARG_MAX}

[2]In a future revision of this part of ISO/IEC 9945, omitting a symbol defined in this subclause from `<limits.h>` is expected to indicate that the value is variable.

Table 2.5—Run-Time Invariant Values (Possibly Indeterminate) (Continued)

Name	Description	Minimum Value
{CHILD_MAX}	Maximum number of simultaneous processes per real user ID.	{_POSIX_CHILD_MAX}
{DELAYTIMER_MAX}	Maximum number of timer expiration overruns.	{_POSIX_DELAYTIMER_MAX}
{LOGIN_NAME_MAX}	Maximum length of a login name.	{_POSIX_LOGIN_NAME_MAX}
{MQ_OPEN_MAX}	The maximum number of open message queue descriptors a process may hold.	{_POSIX_MQ_OPEN_MAX}
{MQ_PRIO_MAX}	The maximum number of message priorities supported by the implementation.	{_POSIX_MQ_PRIO_MAX}
{OPEN_MAX}	Maximum number of files that one process can have open at any given time.	{_POSIX_OPEN_MAX}
{PAGESIZE}	Granularity in bytes of memory mapping and process memory locking.	1
{PTHREAD_DESTRUCTOR_-ITERATIONS}	Maximum number of attempts made to destroy the thread-specific data values of a thread on thread exit.	{_POSIX_THREAD_DESTRUCTOR_-ITERATIONS}
{PTHREAD_KEYS_MAX}	Maximum number of data keys that can be created per process.	{_POSIX_THREAD_KEYS_MAX}
{PTHREAD_STACK_MIN}	Mininum size in bytes of thread stack storage.	0
{PTHREAD_THREADS_MAX}	Maximum number of threads that can be created per process.	{_POSIX_THREAD_THREADS_MAX}
{RTSIG_MAX}	Maximum number of real time signals reserved for application use in this implementation.	{_POSIX_RTSIG_MAX}
{SEM_NSEMS_MAX}	Maximum number of semaphores that a process may have.	{_POSIX_SEM_NSEMS_MAX}
{SEM_VALUE_MAX}	The maximum value a semaphore may have.	{_POSIX_SEM_VALUE_MAX}
{SIGQUEUE_MAX}	Maximum number of queued signals that a process may send and have pending at the receiver(s) at any time.	{_POSIX_SIGQUEUE_MAX}
{STREAM_MAX}	The number of streams that one process can have open at one time. If defined, it shall have the same value as {FOPEN_MAX} from the C Standard {2}.	{_POSIX_STREAM_MAX}
{TIMER_MAX}	Maximum number of timers per process supported by the implementation.	{_POSIX_TIMER_MAX}
{TTY_NAME_MAX}	Maximum length of terminal device name.	{_POSIX_TTY_NAME_MAX}
{TZNAME_MAX}	The maximum number of bytes supported for the name of a time zone (not of the **TZ** variable).	{_POSIX_TZNAME_MAX}

Table 2.6—Pathname Variable Values

Name	Description	Minimum Value
{LINK_MAX}	Maximum value of a file's link count.	{_POSIX_LINK_MAX}
{MAX_CANON}	Maximum number of bytes in a terminal canonical input line. (See 7.1.1.6.)	{_POSIX_MAX_CANON}
{MAX_INPUT}	Minimum number of bytes for which space will be available in a terminal input queue; therefore, the maximum number of bytes a portable application may require to be typed as input before reading them.	{_POSIX_MAX_INPUT}
{NAME_MAX}	Maximum number of bytes in a file name (not a string length; count excludes a terminating null).	{_POSIX_NAME_MAX}
{PATH_MAX}	Maximum number of bytes in a pathname (not a string length; count excludes a terminating null).	{_POSIX_PATH_MAX}
{PIPE_BUF}	Maximum number of bytes that can be written atomically when writing to a pipe.	{_POSIX_PIPE_BUF}

2.8.6 Invariant Values

The value in Table 2.7 shall not vary in a given implementation. The value in that table shall appear in `<limits.h>`.

Table 2.7—Invariant Value

Name	Description	Value
{SSIZE_MAX}	The maximum value that can be stored in an object of type *ssize_t*.	{_POSIX_SSIZE_MAX}

2.8.7 Maximum Values

The symbols in Table 2-7a shall be defined in `<limits.h>` with the values shown. These are symbolic names for the most restrictive value for certain features on a system conforming to this part of ISO/IEC 9945. A conforming implementation shall provide values no larger than these values. A portable application shall not require a smaller value for correct operation.

Table 2-7a —Maximum Values

Name	Description	Value
{_POSIX_CLOCKRES_MIN}	The CLOCK_REALTIME clock resolution, in nanoseconds	20 000 000

2.9 Symbolic Constants

A conforming implementation shall have the header `<unistd.h>`. This header defines the symbolic constants and structures referenced elsewhere in this part of ISO/IEC 9945. The constants defined by this header are shown in the following subclauses. The actual values of the constants are implementation defined.

2.9.1 Symbolic Constants for the *access*() Function

The constants used by the *access*() function are shown in Table Default. The constants F_OK, R_OK, W_OK, and X_OK, and the expressions

R_OK | W_OK

(where the | represents the bitwise inclusive OR operator),

R_OK | X_OK

and

R_OK | W_OK | X_OK

shall all have distinct values.

Table 2.8—Symbolic Constants for the *access*() Function

Constant	Description
R_OK	Test for read permission.
W_OK	Test for write permission.
X_OK	Test for execute or search permission.
F_OK	Test for existence of file.

2.9.2 Symbolic Constant for the *lseek*() Function

The constants used by the *lseek*() function are shown in Table 2.9.

Table 2.9—Symbolic Constants for the *lseek*() Function

Constant	Description
SEEK_SET	Set file offset to *offset*.
SEEK_CUR	Set file offset to current plus *offset*.
SEEK_END	Set file offset to EOF plus *offset*.

2.9.3 Compile-Time Symbolic Constants for Portability Specifications

The constants in Table 2-10 may be used by the application, at compile time, to determine which optional facilities are present and what actions shall be taken by the implementation.

If the symbol {_POSIX_MEMLOCK_RANGE} is defined, the symbol {_POSIX_MEMLOCK} shall be defined. If the symbol {_POSIX_MEMORY_PROTECTION} is defined, then at least one of the symbols {_POSIX_MAPPED_FILES} or {_POSIX_SHARED_MEMORY_OBJECTS} shall be defined. If the symbol {_POSIX_SYNCHRONIZED_IO} is defined, the symbol {_POSIX_FSYNC} shall be defined.

If the symbol {_POSIX_THREADS} is defined, then the symbol {_POSIX_THREAD_SAFE_FUNCTIONS} shall also be defined.

If the symbol {_POSIX_THREAD_PRIORITY_SCHEDULING} is defined, then the symbol {_POSIX_THREADS} shall also be defined. If the symbol {_POSIX_THREAD_PRIO_INHERIT} is defined, then the symbol {_POSIX_THREAD_PRIORITY_SCHEDULING} shall also be defined. If the symbol{_POSIX_THREAD_-PRIO_PROTECT} is defined, then the symbol {_POSIX_THREAD_PRIORITY_SCHEDULING} shall also be defined.

Although a Strictly Conforming POSIX.1 Application can rely on the values compiled from the `<unistd.h>` header to afford it portability on all instances of an implementation, it may choose, to interrogate a value at run-time to take advantage of the current configuration. See 4.8.1.

Table 2.10—Compile-Time Symbolic Constants

Name	Description
{_POSIX_ASYNCHRONOUS_IO}	If this symbol is defined, the implementation supports the Asynchronous Input and Output option.
{_POSIX_FSYNC}	If this symbol is defined, the implementation supports the File Synchronization option.
{_POSIX_JOB_CONTROL}	If this symbol is defined, it indicates that the implementation supports the Job Control option.
{_POSIX_MAPPED_FILES}	If this symbol is defined, the implementation supports the Memory Mapped Files option.
{_POSIX_MEMLOCK}	If this symbol is defined, the implementation supports the Process Memory Locking option.
{_POSIX_MEMLOCK_RANGE}	If this symbol is defined, the implementation supports the Range Memory Locking option.
{_POSIX_MEMORY_PROTECTION}	If this symbol is defined, the implementation supports the Memory Protection option.
{_POSIX_MESSAGE_PASSING}	If this symbol is defined, the implementation supports the Message Passing option.
{_POSIX_PRIORITIZED_IO}	If this symbol is defined, the implementation supports the Prioritized Input and Output option.
{_POSIX_PRIORITY_SCHEDULING}	If this symbol is defined, the implementation supports the Process Scheduling option.
{_POSIX_REALTIME_SIGNALS}	If this symbol is defined, the implementation supports the Realtime Signals Extension option.
{_POSIX_SAVED_IDS}	If defined, each process has a saved set-user-ID and a saved set-group-ID.
{_POSIX_SEMAPHORES}	If this symbol is defined, the implementation supports the Semaphores option.
{_POSIX_SHARED_MEMORY_OBJEC TS}	If this symbol is defined, the implementation supports the Shared Memory Objects option.
{_POSIX_SYNCHRONIZED_IO}	If this symbol is defined, the implementation supports the Synchronized Input and Output option.
{_POSIX_THREADS}	If this symbol is defined, the implementation supports the Threads option.
{_POSIX_THREAD_ATTR_STACKADDR}	If this symbol is defined, the implementation supports the Thread Stack Address Attribute option.
{_POSIX_THREAD_ATTR_STACKSIZE}	If this symbol is defined, the implementation supports the Thread Stack Size Attribute option.
{_POSIX_THREAD_PRIORITY_SCHEDULING}	If this symbol is defined, the implementation supports the Thread Execution Scheduling option.

Table 2.10—Compile-Time Symbolic Constants (Continued)

Name	Description
{_POSIX_THREAD_PRIO_INHERIT}	If this symbol is defined, the implementation supports the Priority Inheritance option.
{_POSIX_THREAD_PRIO_PROTECT}	If this symbol is defined, the implementation supports the Priority Protection option.
{_POSIX_THREAD_PROCESS_SHARED}	If this symbol is defined, the implementation supports the Process-Shared Synchronization option.
{_POSIX_THREAD_SAFE_FUNCTIONS}	If this symbol is defined, the implementation supports the Thread-Safe Functions option.
{_POSIX_TIMERS}	If this symbol is defined, the implementation supports the Timers option.
{_POSIX_VERSION}	The integer value 199506L. This value shall be used for system that conform to this part of ISO/IEC 9945.

2.9.4 Execution-Time Symbolic Constants for Portability Specifications

The constants in Table 2-11 may be used by the application, at execution time, to determine which optional facilities are present and what actions shall be taken by the implementation in some circumstances described by this part of ISO/IEC 9945 as *implementation defined.*

Table 2-11 —Execution-Time Symbolic Constants

Name	Description
{_POSIX_ASYNC_IO}	Asynchronous input or output operations may be performed for the associated file.
{_POSIX_CHOWN_RESTRICTED}	The implementation supports the Change File Owner Restriction. The use of the *chown*() function is restricted to a process with appropriate privileges, and to changing the group ID of a file only to the effective group ID of the process or to one of its supplementary group IDs.
{_POSIX_NO_TRUNC}	Pathname components longer than {NAME_MAX} generate an error.
{_POSIX_PRIO_IO}	Prioritized input or output operations may be performed for the associated file.
{_POSIX_SYNC_IO}	Synchronized input or output operations may be performed for the associated file.
{_POSIX_VDISABLE}	Terminal special characters defined in 7.1.1.9 can be disabled using this character value, if it is defined. See *tcgetattr*() and *tcsetattr*().

If any of the constants in Table 2-11 are not defined in the header `<unistd.h>`, the value varies depending on the file to which it is applied. See 5.7.1.

If any of the constants in Table 2-11 are defined to have value –1 in the header `<unistd.h>`, the implementation shall not provide the option on any file; if any are defined to have a value other than –1 in the header `<unistd.h>`, the implementation shall provide the option on all applicable files.

All of the constants in Table 2-11, whether defined in `<unistd.h>` or not, may be queried with respect to a specific file using the *pathconf*() or *fpathconf*() functions.

3. Process Primitives

The functions described in this section perform the most primitive operating system services dealing with processes, interprocess signals, and timers. All attributes of a process that are specified in this part of ISO/IEC 9945 shall remain unchanged by a process primitive unless the description of that process primitive states explicitly that the attribute is changed.

3.1 Process Creation and Execution

3.1.1 Process Creation

Function: *fork*()

3.1.1.1 Synopsis

```
#include <sys/types.h>
pid_t fork(void);
```

3.1.1.2 Description

The *fork*() function creates a new process. The new process (child process) shall be an exact copy of the calling process (parent process) except for the following:

1) The child process has a unique process ID. The child process ID also does not match any active process group ID.
2) The child process has a different parent process ID (which is the process ID of the parent process).
3) The child process has its own copy of the parent's file descriptors. Each of the child's file descriptors refers to the same open file description with the corresponding file descriptor of the parent.
4) The child process has its own copy of the parent's open directory streams (see 5.1.2). Each open directory stream in the child process may share directory stream positioning with the corresponding directory stream of the parent.
5) The child process's values of *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* are set to zero (see 4.5.2).
6) File locks previously set by the parent are not inherited by the child.(See 6.5.2.)
7) Pending alarms are cleared for the child process. (See 3.4.1.)
8) The set of signals pending for the child process is initialized to the empty set. (See 3.3.1.1.)
9) If the Semaphores option is supported, any semaphores that are open in the parent process shall also be open in the child process. (See 11.2.)
10) If the Process Memory Locking option is supported, the child process shall not inherit any address space memory locks established by the parent process via calls to *mlockall*() or *mlock*(). (See 12.1.)
11) If the Memory Mapped Files or Shared Memory Objects option is supported, memory mappings created in the parent are retained in the child process. MAP_PRIVATE mappings inherited from the parent shall also be MAP_PRIVATE mappings in the child, and any modifications to the data in these mappings made by the parent prior to calling *fork*() shall be visible to the child. Any modifications to the data in MAP_PRIVATE mappings made by the parent after *fork*() returns shall be visible only to the parent. Modifications to the data in MAP_PRIVATE mappings made by the child shall be visible only to the child. (See 12.2.)
12) If the Process Scheduling option is supported, for the SCHED_FIFO and SCHED_RR scheduling policies the child process shall inherit the policy and priority settings of the parent process during a *fork*() function. For other scheduling policies, the policy and priority settings on *fork*() are implementation defined. (See 13.3.)
13) If the Timers option is supported, per-process timers created by the parent are not inherited by the child process. (See Section 14)

14) If the Message Passing option is supported, the child process has its own copy of the message queue descriptors of the parent. Each of the message descriptors of the child refers to the same open message queue description as the corresponding message descriptor of the parent. (See Section 15)
15) If the Asynchronous Input and Output option is supported, no asynchronous input or asynchronous output operations are inherited by the child process. (See 6.7.)
16) A process is created with a single thread. If a multithreaded 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 (see 3.3.1.3) until such time as one of the exec functions is called. If {_POSIX_THREADS} is defined, fork handlers may be established by means of the *pthread_atfork*() function in order to maintain application invariants across *fork*() calls.

All other process characteristics defined by this part of ISO/IEC 9945 shall be the same in the parent and the child processes. The inheritance of process characteristics not defined by this part of ISO/IEC 9945 is unspecified by this part of ISO/IEC 9945, but should be documented in the system documentation.

After *fork*(), both the parent and the child processes shall be capable of executing independently before either terminates.

3.1.1.3 Returns

Upon successful completion, *fork*() shall return a value of zero 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, a value of –1 shall be returned to the parent process, no child process shall be created, and *errno* shall be set to indicate the error.

3.1.1.4 Errors

If any of the following conditions occur, the *fork*() function shall return –1 and set *errno* to the corresponding value:

[EAGAIN]	The system lacked the necessary resources to create another process, or the system-imposed limit on the total number of processes under execution by a single user would be exceeded.

For each of the following conditions, if the condition is detected, the *fork*() function shall return –1 and set *errno* to the corresponding value:

[ENOMEM]	The process requires more space than the system is able to supply.

3.1.1.5 Cross-References

alarm(), 3.4.1; *exec*, 3.1.2; *fcntl*(), 6.5.2; *kill*(), 3.3.2; *times*(), 4.5.2; *wait*, 3.2.1.

3.1.2 Execute a File

Functions: *execl*(), *execv*(), *execle*(), *execve*(), *execlp*(), *execvp*().

3.1.2.1 Synopsis

```
int execl(const char *path, const char *arg, ...);
int execv(const char *path, char *const argv[]);
int execle(const char *path, const char *arg, ...);
int execve(const char *path, char *const argv[], char *cons envp[]);
int execlp(const char *file, const char *arg, ...);
int execvp(const char *file, char *const argv[]);
```

3.1.2.2 Description

The *exec* family of functions shall replace the current process image with a new process image. The new image is constructed from a regular, executable file called the *new process image file*. There shall be no return from a successful *exec* because the calling process image is overlaid by the new process image.

When a C program is executed as a result of this call, it shall be entered as a C language function call as follows:

```
int main(int argc, char *argv[]);
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. In addition, the following variable:

```
extern char **environ;
```

is initialized as a pointer to an array of character pointers to the environment strings. The *argv* and *environ* arrays are each terminated by a **NULL** pointer. The **NULL** pointer terminating the *argv* array is not counted in *argc.*

The arguments specified by a program with one of the *exec* functions shall be passed on to the new process image in the corresponding *main*() arguments.

The argument *path* points to a pathname that identifies the new process image file.

The argument *file* is used to construct a pathname that identifies the new process image file. If the *file* argument contains a slash character, the *file* argument shall be used as the pathname for this file. Otherwise, the path prefix for this file is obtained by a search of the directories passed as the environment variable **PATH** (see 2.6). If this environment variable is not present, the results of the search are implementation defined.

The argument *argv* is an array of character pointers to null-terminated strings. The last member of this array shall be a **NULL** pointer. These strings constitute the argument list available to the new process image. The value in *argv*[0] should point to a filename that is associated with the process being started by one of the *exec* functions.

The `const char` **arg* and subsequent ellipses in the *execl*(), *execlp*(), and *execle*() functions can be thought of as *arg0*, *arg1*, …, *argn*. Together they describe a list of one or more pointers to null-terminated character strings that represent the argument list available to the new program. The first argument should point to a filename that is associated with the process being started by one of the *exec* functions, and the last argument shall be a **NULL**. pointer. For the *execle*() function, the environment is provided by following the **NULL** pointer that shall terminate the list of arguments in the parameter list to *execle*() with an additional parameter, as if it were declared as

```
char *const envp[]
```

The argument *envp* to *execve*() and the final argument to *execle*() name an array of character pointers to null-terminated strings. These strings constitute the environment for the new process image. The environment array is terminated by a **NULL** pointer.

For those forms not containing an *envp* pointer [*execl*(), *execv*(), *execlp*(), and *execvp*()], the environment for the new process image is taken from the external variable *environ* in the calling process.

The number of bytes available for the new process's combined argument and environment lists is {ARG_MAX}. The implementation shall specify in the system documentation (see 1.3.1.2) whether any combination of null terminators, pointers, or alignment bytes are included in this total.

If {_POSIX_THREADS} is defined, conforming multithreaded applications shall not use the *environ* variable to access or modify any environment variable while any other thread is concurrently modifying any environment variable. A call to any function dependent on any environment variable shall be considered a use of the *environ* variable to access that environment variable.

NOTE — Functions for setting and clearing environment variables are currently proposed. When such functions are standardized, they will be defined to be safe to use for setting and clearing environment variables in multithreaded programs.

File descriptors open in the calling process image remain open in the new process image, except for those whose close-on-exec flag FD_CLOEXEC is set (see 6.5.2 and 6.5.1). For those file descriptors that remain open, all attributes of the open file description, including file locks (see 6.5.2), remain unchanged by this function call.

Directory streams open in the calling process image shall be closed in the new process image.

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. 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 3.3.1.1).

If the set-user-ID mode bit of the new process image file is set (see 5.6.4), the effective user ID of the new process image is set to the owner 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 is 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 remain the same as those of the calling process image. If {_POSIX_SAVED_IDS} is defined, 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.

If the Semaphores option is supported, any named semaphores that are open in the calling process shall be closed as if by appropriate calls to *sem_close*().

If the Process Memory Locking option is supported, 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.

If the Memory Mapped Files or Shared Memory Objects option is supported, memory mappings created in the process are unmapped before the address space is rebuilt for the new process image.

If the Process Scheduling option is supported, for the SCHED_FIFO and SCHED_RR scheduling policies the policy and priority settings shall not be changed by a call to an *exec* function. For other scheduling policies, the policy and priority settings on *exec* are implementation defined.

If the Timers option is supported, per-process timers created by the calling process shall be deleted before replacing the current process image with the new process image.

If the Message Passing option is supported, all open message queue descriptors in the calling process shall be closed, as described in *mq_close*().

If the Asynchronous Input and Output option is supported, 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 cancelled, and which I/O may be cancelled upon *exec,* is implementation defined.

A call to any *exec* function from a process with more than one thread shall result in all threads being terminated and the new executable image being loaded and executed. No destructor functions shall be called.

The new process image also inherits the following attributes from the calling process image:

1) Process ID
2) Parent process ID
3) Process group ID
4) Session membership
5) Real user ID
6) Real group ID
7) Supplementary group IDs
8) Time left until an alarm clock signal (see 3.4.1)
9) Current working directory
10) Root directory
11) File mode creation mask (see 5.3.3)
12) Process signal mask (see 3.3.5)
13) Pending signals (see 3.3.6)
14) *tms_utime, tms_stime, tms_cutime,* and *tms_cstime* (see 4.5.2)

All process attributes defined by this part of ISO/IEC 9945 and not specified in this subclause (3.1.2) shall be the same in the new and old process images. The inheritance of process characteristics not defined by this part of ISO/IEC 9945 is unspecified by this part of ISO/IEC 9945, but should be documented in the system documentation.

Upon successful completion, the *exec* functions shall mark for update the *st_atime* field of the file. If the *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 *open*()-ed. 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.

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.

3.1.2.3 Returns

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.

3.1.2.4 Errors

If any of the following conditions occur, the *exec* functions shall return –1 and set *errno* to the corresponding value:

[E2BIG]	The number of bytes used by the argument list and the environment list of the new process image is greater than the system-imposed limit of {ARG_MAX} bytes.
[EACCES]	Search permission is denied for a directory listed in the path prefix of the new process image file, 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.
[ENAMETOOLONG]	The length of the *path* or *file* arguments, or an element of the environment variable **PATH** prefixed to a file, exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} and {_POSIX_NO_TRUNC} is in effect for that file.

[ENOENT]	One or more components of the pathname of the new process image file do not exist, or the *path* or *file* argument points to an empty string.
[ENOTDIR]	A component of the path prefix of the new process image file is not a directory.

If any of the following conditions occur, the *execl*(), *execv*(), *execle*(), and *execve*() functions shall return −1 and set *errno* to the corresponding value:

[ENOEXEC]	The new process image file has the appropriate access permission, but is not in the proper format.

For each of the following conditions, if the condition is detected, the *exec* functions shall return −1 and return the corresponding value in *errno*:

[ENOMEM]	The new process image requires more memory than is allowed by the hardware or system-imposed memory management constraints.

3.1.2.5 Cross-References

alarm(), 3.4.1; *chmod*(), 5.6.4;*_exit*(), 3.2.2; *fcntl*(), 6.5.2; *fork*(), 3.1.1; *setuid*(), 4.2.2; `<signal.h>`, 3.3.1.1; *sigprocmask*(), 3.3.5; *sigpending*(), 3.3.6; *stat*(), 5.6.2; `<sys/stat.h>`, 5.6.1; *times*(), 4.5.2; *umask*(), 5.3.3; 2.6.

3.1.3 Register Fork Handlers

Function: *pthread_atfork*()

3.1.3.1 Synopsis

```
#include <sys/types.h>
int pthread_atfork(void (*prepare) (void), void (*parent) (void),
            void (*child) (void));
```

3.1.3.2 Description

If {_POSIX_THREADS} is defined:

> 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 handler shall be called after *fork*() processing completes in the parent process. The *child* fork handler shall be called alter *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.

Otherwise:

> Either the implementation shall support the *pthread_atfork*() function as described above or the *pthread_atfork*() function shall not be provided.

3.1.3.3 Returns

Upon successful completion, *pthread_atfork*() shall return a value of zero. Otherwise, an error number shall be returned to indicate the error.

3.1.3.4 Errors

If any of the following conditions occur, the *pthread_atfork*() function shall return the corresponding error number:

[ENOMEM]	Insufficient table space exists to record the fork handler addresses.

3.1.3.5 Cross-References

fork(), 3.1.1.

3.2 Process Termination

There are two kinds of process termination:

1) *Normal termination* occurs by a return from *main*() or when requested with the *exit*() or *_exit*() functions.
2) *Abnormal termination* occurs when requested by the *abort*() function or when some signals are received (see 3.3.1.1).

The *exit*() and *abort*() functions shall be as described in the C Standard {2}. Both *exit*() and *abort*() shall terminate a process with the consequences specified in 3.2.2, except that the status made available to *wait*() or *waitpid*() by *abort*() shall be that of a process terminated by the SIGABRT signal.

A thread of the parent process can suspend its execution to wait for termination of a child process with the *wait*() or *waitpid*() functions.

3.2.1 Wait for Process Termination

Functions: *wait*(), *waitpid*()

3.2.1.1 Synopsis

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

3.2.1.2 Description

The *wait*() and *waitpid*() functions allow the calling process to obtain status information pertaining to one of its 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 a signal whose action is either to execute a signal-catching function tion or to terminate the process is delivered. 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 behave identically to the *wait*() function if the pid argument has a value of −1 and the *options* argument has a value of zero. Other wise, 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.

1) If *pid* is equal to −1, status is requested for any child process. In this respect, *waitpid*() is then equivalent to *wait*().
2) If *pid* is greater than zero, it specifies the process ID of a single child process for which status is requested.
3) If *pid* is equal to zero, status is requested for any child process whose process group ID is equal to that of the calling process.
4) If *pid* is less than −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 header `<sys/wait.h>`:

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

If the implementation supports Job Control, the status of any child processes specified by *pid* that are stopped, and whose status has not yet been reported since they stopped, shall also be reported to the requesting process.

If *wait*() or *waitpid*() return because the status of a child process is available, these functions shall return a value equal to the process ID of the child process. In this case, if the value of the argument *stat_loc* is not **NULL**, information shall be stored in the location pointed to by *stat_loc*. If and only if the status returned is from a terminated child process that returned a value of zero from *main*() or passed a value of zero as the *status* argument to *_exit*() or *exit*(), the value stored at the location pointed to by *stat_loc* shall be zero. Regardless of its value, this information may be interpreted using the following macros, which are defined in `<sys/wait.h>` and evaluate to integral expressions; the *stat_val* argument is the integer value pointed to by *stat_loc*.

WIFEXITED(*stat_val*)

This macro evaluates to a nonzero value if status was returned for a child process that terminated normally.

WEXITSTATUS(*stat_val*)

If the value of WIFEXITED(*stat_val*) is nonzero, this macro evaluates to the low-order 8 bits of the *status* argument that the child process passed to *_exit*() or *exit*(), or the value the child process returned from *main*().

WIFSIGNALED(*stat_val*)

This macro evaluates to a nonzero value if status was returned for a child process that terminated due to the receipt of a signal that was not caught (see 3.3.1.1).

WTERMSIG(*stat_val*)

If the value of WIFSIGNALED(*stat_val*) is nonzero, this macro evaluates to the number of the signal that caused the termination of the child process.

WIFSTOPPED(*stat_val*)

This macro evaluates to a nonzero value if status was returned for a child process that is currently stopped.

WSTOPSIG(*stat_val*)

If the value of WIFSTOPPED(*stat_val*) is nonzero, this macro evaluates to the number of the signal that caused the child process to stop.

If the information stored at the location pointed to by *stat_loc* was stored there by a call to the *waitpid*() function that specified the WUNTRACED flag, exactly one of the macros WIFEXITED(**stat_loc*), WIFSIGNALED(**stat_loc*), or WIFSTOPPED(**stat_loc*) shall evaluate to a nonzero value. If the information stored at the location pointed to by *stat_loc* was stored there by a call to the *wait-pid*() function that did not specify the WUNTRACED flag or by a call to the *wait*() function, exactly one of the macros WIFEXITED(**stat_loc*) or WIFSIGNALED(**stat_loc*) shall evaluate to a nonzero value.

An implementation may define additional circumstances under which *wait*() or *waitpid*() reports status. This shall not occur unless the calling process or one of its child processes explicitly makes use of a nonstandard extension. In these cases, the interpretation of the reported status is implementation defined.

3.2.1.3 Returns

If the *wait*() or *waitpid*() functions return because the status of a child process is available, these functions shall return a value equal to the process ID of the child process for which status is reported. If the *wait*() or *waitpid*() functions return due to the delivery of a signal to the calling process, a value of –1 shall be returned and *errno* shall be set to [EINTR]. If the *waitpid*() function was invoked with WNOHANG set in *options*, 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*, a value of zero shall be returned. Otherwise, a value of –1 shall be returned, and *errno* shall be set to indicate the error.

3.2.1.4 Errors

If any of the following conditions occur, the *wait*() function shall return –1 and set *errno* to the corresponding value:

[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.

If any of the following conditions occur, the *waitpid*() function shall return –1 and set *errno* to the corresponding value:

[ECHILD]	The process or process group specified by *pid* does not exist or is not 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 value of the *options* argument is not valid.

3.2.1.5 Cross-References

_exit(), 3.2.2; *fork*(), 3.1.1; *pause*(), 3.4.2; *times*(), 4.5.2; `<signal.h>`, 3.3.1.1.

3.2.2 Terminate a Process

Function:*_exit*()

3.2.2.1 Synopsis

```
void _exit(int status);
```

3.2.2.2 Description

The *_exit*() function shall terminate the calling process with the following consequences:

1) All open file descriptors and directory streams in the calling process are closed.
2) If the parent process of the calling process is executing a *wait*() or *waitpid*(), it is notified of the termination of the calling process and the low order 8 bits of *status* are made available to it; see 3.2.1.
3) If the parent process of the calling process is not executing a *wait*() or *waitpid*() function, the exit *status* code is saved for return to the parent process whenever the parent process executes an appropriate subsequent *wait*() or *waitpid*().
4) 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. Children of a terminated process shall be assigned a new parent process ID, corresponding to an implementation-defined system process.
5) If the implementation supports the SIGCHLD signal, a SIGCHLD signal shall be sent to the parent process.
6) 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.
7) If the process is a controlling process, the controlling terminal associated with the session is disassociated from the session, allowing it to be acquired by a new controlling process.
8) If the implementation supports job control, and 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.
9) If the Semaphores option is supported, all open named semaphores in the calling process shall be closed as if by appropriate calls to *sem_close*().
10) If the Process Memory Locking option is supported, 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*().
11) If the Memory Mapped Files or Shared Memory Objects option is supported, memory mappings created in the process are unmapped before the process is destroyed.
12) If the Message Passing option is supported, all open message queue descriptors in the calling process shall be closed as if by appropriate calls to *mq_close*().
13) If the Asynchronous Input and Output option is supported, 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*() operation held not yet occurred, but any associated signal notifications shall be suppressed. The *_exit*() operation itself may or may not block awaiting such I/O completion. Whether any I/O is cancelled, and which I/O may be cancelled upon *_exit*(), is implementation defined.
14) Threads terminated by a call to *_exit*() shall not invoke their cancellation cleanup handlers (see Section 18) and shall not invoke per-thread data destructors (see Section 17).

These consequences shall occur on process termination for any reason.

3.2.2.3 Returns

The *_exit*() function cannot return to its caller.

3.2.2.4 Cross-References

close(), 6.3.1; *sigaction*(), 3.3.4; *wait*, 3.2.1.

3.3 Signals

3.3.1 Signal Concepts

3.3.1.1 Signal Names

The `<signal.h>` header declares the *sigset_t* type and the *sigaction* structure. It also defines the following symbolic constants, each of which expands to a distinct constant expression of the type *void(*)*(), whose value matches no declarable function.

Symbolic Constant	**Description**
SIG_DFL	Request for default signal handling
SIG_IGN	Request that signal be ignored

The type *sigset_t* is used to represent sets of signals. It is always an integral or structure type. Several functions used to manipulate objects of type *sigset_t* are defined in 3.3.3.2.

The `<signal.h>` header also declares the constants that are used to refer to the signals that occur in the system. Each of the signals defined by this part of ISO/IEC 9945 and supported by the implementation shall have distinct, positive integral values. The value zero is reserved for use as the null signal (see 3.3.2). An implementation may define additional signals that may occur in the system.

The constants shown in Table 3.1 shall be supported by all implementations.

The constants shown in Table 3.2 and Table 3.3 shall be defined by all implementations. However, implementations that do not support Job Control are not required to support the signals in Table 3.2, and those not supporting Memory Protection are not required to support the signals in Table 3.3. If these signals are supported by the implementation, they shall behave in accordance with this part of ISO/IEC 9945. Otherwise, the implementation shall not generate these signals, and attempts to send these signals or to examine or specify their actions shall return an error condition. See 3.3.2 and 3.3.4.

Table 3.1—Required Signals

Symbolic Constant	Default Action	Description
SIGABRT	1	Abnormal termination signal, such as is initiated by the *abort*() function (as defined in the C Standard {2}).
SIGALRM	1	Timeout signal, such as initiated by the *alarm*() function (see 3.4.1).
SIGFPE	1	Erroneous arithmetic operation, such as division by zero or an operation resulting in overflow.
SIGHUP	1	Hangup detected on controlling terminal (see 7.1.1.10) or death of controlling process (see 3.2.2).
SIGILL	1	Detection of an invalid hardware instruction.
SIGINT	1	Interactive attention signal (see 7.1.1.9).
SIGKILL	1	Termination signal (cannot be caught or ignored).
SIGPIPE	1	Write on a pipe with no readers (see 6.4.2).
SIGQUIT	1	Interactive termination signal (see 7.1.1.9).
SIGSEGV	1	Detection of an invalid memory reference.
SIGTERM	1	Termination signal.
SIGUSR1	1	Reserved as application-defined signal 1.
SIGUSR2	1	Reserved as application-defined signal 2.

NOTE — The default actions are

1	Abnormal termination of the process.

Table 3.2—Job Control Signals

Symbolic Constant	Default Action	Description
SIGCHLD	2	Child process terminated or stopped.
SIGCONT	4	Continue if stopped.
SIGSTOP	3	Stop signal (cannot be caught or ignored).
SIGTSTP	3	Interactive stop signal (see 7.1.1.9).
SIGTTIN	3	Read from control terminal attempted by a member of a background process group (see 7.1.1.4).
SIGTTOU	3	Write to control terminal attempted by a member of a background process group (see 7.1.1.4).

NOTE — The default actions are

2	Ignore the signal.
3	Stop the process.
4	Continue the process if it is currently stopped; otherwise, ignore the signal.

The macros SIGRTMIN and SIGRTMAX shall be defined in `<signal.h>`, shall evaluate to integral expressions, and, if the Realtime Signals Extension option is supported, shall specify a range of signal numbers that are reserved for application use and for which the realtime signal behavior specified in this section shall be supported. The signal numbers in this range shall not overlap any of the signals specified in Tables 3.1, 3.2, or 3.3.

Table 3.3—Memory Protection Signals

Symbolic Constant	Default Action	Description
SIGBUS	1	Access to an undefined portion of a memory object (see 5.6.7, 12.2.1).

The range SIGRTMIN through SIGRTMAX inclusive shall include at least {RTSIG_MAX} signal numbers.

It is implementation defined whether the realtime signal behavior specified in this section—specifically, the queuing of signals and the passing of application-defined values—is supported for the signals defined in Tables 3.1, 3.2, or 3.3.

3.3.1.2 Signal Generation and Delivery

A signal is said to be *generated* for (or sent to) a process or a 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 that 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 3.3.1.3). 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 being delivered 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 a thread, the signal shall remain pending until either it is accepted, it is unblocked, or the action associated with it is set to ignore the signal. Signals generated for the process shall be delivered to or accepted by exactly one of those threads within the process. The thread shall be in a call to a *sigwait* function selecting that signal or it shall not block 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 either 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 or the thread, 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 is initialized from that of its 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 *sigaction*(), *sigprocmask*(), *pthread_sigmask*(), 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 for which queueing is required under the Realtime Signals Extension option. The order in which multiple, simultaneously pending signals outside the range SIGRTMIN to SIGRTMAX are delivered to or accepted by a process is unspecified.

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

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.

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 shall be defined in `<signal.h>` and shall contain at least the following members:

Member Type	**Name Member**	**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

Implementations may add extensions as permitted in 1.3.1.1, item (2). Adding extensions to this structure, which may change the behavior of the application with respect to this standard when those fields in the structure are uninitialized, also requires that the extension be enabled as required by 1.3.1.1.

The *sigev_notify* member specifies the notification mechanism to use when an asynchronous event occurs. This standard 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.

Multithreaded programs can use an alternate event notification mechanism. If {_POSIX_THREADS} is defined:

> When a notification is processed, and when the *sigev_notify* member of the *sigevent* structure has the value SIGEV_THREAD, the function *sigev_notify_function* shall be called with the 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 a thread attributes object with a `detachstate` attribute of PTHREAD_CREATE_JOINABLE results in undefined behavior. The signal mask of this thread is implementation defined.

Otherwise:

> Either the implementation shall support the behavior specified above or the implementation shall treat the SIGEV_THREAD value as an error.

The *sigval* union shall be defined in `<signal.h>` and shall contain 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.

If the Realtime Signals Extension option is supported:

> 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 shall be 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.

An implementation shall document any conditions not specified by this part of ISO/IEC 9945 under which the implementation generates signals. (See 1.3.1.2.)

3.3.1.3 Signal Actions

There are three types of actions 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 3.1.2). The actions prescribed by these values are as follows:

1) SIG_DFL — signal-specific default action
   - a) The default actions for the signals defined in this part of ISO/IEC 9945 are specified in Tables 3.1, 3.2, and 3.3. If the Realtime Signals Extension option is supported, the default actions for the realtime signals in the range of SIGRTMIN through SIGRTMAX shall be to terminate the process abnormally.
   - b) 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 (see 3.3.4). 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.
   - c) 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. If the Realtime Signals Extension option is supported, any queued values pending shall be discarded, and the resources used to queue them shall be released and made available to queue other signals.
2) SIG_IGN — ignore signal
   - a) 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 the *kill*() function, the *sigqueue*() function, or the *raise*() function as defined by the C Standard {2}.
   - b) The system shall not allow the action for the signals SIGKILL or SIGSTOP to be set to SIG_IGN.
   - c) 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. Any queued values pending shall be discarded, and the resources used to queue them shall be released and made available to queue other signals.
   - d) If a process sets the action for the SIGCHLD signal to SIG_IGN, the behavior is unspecified.
3) *pointer to a function* — catch signal
   - a) 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.
   - b) 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 and *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 member(s):

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 as defined in the C Standard {2} or any similar functions provided as implementation extensions.
SI_QUEUE	The signal was sent by the *sigqueue*() function.
SI_TIMER	The signal was generated by the expiration of a timer set by *timer_settime*().
SI_ASYNCIO	The signal was generated by the completion of an asynchronous I/O request.
SI_MESGQ	The signal was generated by the arrival of a message on an empty message queue.

If the signal was not generated by one of the functions or events listed above, the *si_code* shall be set to an implementation-defined value that is not equal to any of the values defined above.
If the Realtime Signals Extension option is supported and *si_code* is one of SI_QUEUE, SI_TIMER, SI_ASYNCIO, or SI_MESGQ, then *si_value* shall contain the application-specified signal value. Otherwise, the contents of *si_value* are undefined.
The parameter *context* is undefined by this part of ISO/IEC 9945.

c) The behavior of a process is undefined after it returns normally from a signal-catching function for a SIGFPE, SIGILL, SIGSEGV, or SIGBUS signal that was not generated by the *kill*() function, the *sigqueue*() function, or the *raise*() function as defined by the C Standard {2}.
d) The system shall not allow a process to catch the signals SIGKILL and SIGSTOP.
e) 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.
f) When signal-catching functions are invoked asynchronously with process execution, the behavior of some of the functions defined by this part of ISO/IEC 9945 is unspecified if they are called from a signal-catching function. The following table defines a set of functions that shall be async-signal safe.

access()	*fdatasync*()	*read*()	*tcdrain*()
aio_error()	*fork*()	*rename*()	*tcflow*()
aio_return()	*fstat*()	*rmdir*()	*tcflush*()
aio_suspend()	*fsync*()	*sem_post*()	*tcgetattr*()
alarm()	*getegid*()	*setgid*()	*tcgetpgrp*()
cfgetispeed()	*geteuid*()	*setpgid*()	*tcsendbreak*()
cfgetospeed()	*getgid*()	*setsid*()	*tcsetattr*()
cfsetispeed()	*getgroups*()	*setuid*()	*tcsetpgrp*()
cfsetospeed()	*getpgrp*()	*sigaction*()	*time*()
chdir()	*getpid*()	*sigaddset*()	*timer_getoverrun*()
chmod()	*getppid*()	*sigdelset*()	*timer_gettime*()
chown()	*getuid*()	*sigemptyset*()	*timer_settime*()
clock_gettime()	*kill*()	*sigfillset*()	*times*()
close()	*link*()	*sigismember*()	*umask*()
creat()	*lseek*()	*sigpending*()	*uname*()
dup2()	*mkdir*()	*sigprocmask*()	*unlink*()
dup()	*mkfifo*()	*sigqueue*()	*utime*()
execle()	*open*()	*sigsuspend*()	*wait*()
execve()	*pathconf*()	*sleep*()	*waitpid*()
_exit()	*pause*()	*stat*()	*write*()
fcntl()	*pipe*()	*sysconf*()	

All POSIX.1 functions not in the preceding table and all functions defined in the C Standard {2} not stated to be callable from a signal-catching function are considered to be *unsafe* with respect to signals. In the presence of signals, all functions defined by this part of ISO/IEC 9945 or by the C Standard {2} shall behave as defined (by the defining standard) 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.

3.3.1.4 Signal Effects on Other Functions

Signals affect the behavior of certain functions defined by this part of ISO/IEC 9945 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 causes the process to be continued, and the original function shall continue at the point where 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`, the behavior of the interrupted function shall be as described individually for that function. 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 delivered, except as specified for *sigpending*() (3.3.6) and the *sigwait* functions (3.3.8).

3.3.2 Send a Signal to a Process

Function: *kill*()

3.3.2.1 Synopsis

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

3.3.2.2 Description

The *kill*() function shall send a signal to a process or a group of processes specified by *pid*. The signal to be sent is specified by *sig* and is either one from the list given in 3.3.1.1 or zero. If *sig* 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*.

For a process to have permission to send a signal to a process designated by *pid*, the real or effective user ID of the sending process must match the real or effective user ID of the receiving process, unless the sending process has appropriate privileges. If {_POSIX_SAVED_IDS} is defined, the saved set-user-ID of the receiving process shall be checked in place of its effective user ID.

If *pid* is greater than zero, sig shall be sent to the process whose process ID is equal to *pid*.

If *pid* is zero, *sig* shall be sent to all processes (excluding an unspecified set of system processes) whose process group ID is equal to the process group ID of the sender and for which the process has permission to send a signal.

If *pid* is –1, the behavior of the *kill*() function is unspecified.

If *pid* is negative, but not –1, *sig* shall be sent to all processes (excluding an unspecified set of system processes) whose process group ID is equal to the absolute value of *pid* and for which the process has permission to send a signal.

If the value of *pid* causes *sig* to be generated for the sending process, and if *sig* is not blocked for the calling thread and if no other thread has sig unblocked or is waiting in a *sigwait* function for *sig*, either *sig* or at least one pending unblocked signal shall be delivered to the calling thread before the *kill*() function returns.

If the implementation supports the SIGCONT signal, the user ID tests described above shall not be applied when sending SIGCONT to a process that is a member of the same session as the sending process.

An implementation that provides extended security controls may impose further implementation-defined restrictions on the sending of signals, including the null signal. In particular, the system may deny the existence of some or all of the processes specified by *pid*.

The *kill*() function is successful if the process has permission to send sig to any of the processes specified by *pid*. If the *kill*() function fails, no signal shall be sent.

3.3.2.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of –1 shall be returned and *errno* shall be set to indicate the error.

3.3.2.4 Errors

If any of the following conditions occur, the *kill*() function shall return –1 and set *errno* to the corresponding value:

[EINVAL]	The value of the *sig* argument is an invalid or unsupported signal number.
[EPERM]	The process does not have permission to send the signal to any receiving process.
[ESRCH]	No process or process group can be found corresponding to that specified by *pid*.

3.3.2.5 Cross-References

getpid(), 4.1.1; *setsid*(), 4.3.2; *sigaction*(), 3.3.4; `<signal.h>`, 3.3.1.1.

3.3.3 Manipulate Signal Sets

Functions: *sigemptyset*(), *sigfillset*(), *sigaddset*(), *sigdelset*(), *sigismember*()

3.3.3.1 Synopsis

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(const sigset_t *set, int signo);
```

3.3.3.2 Description

The *sigsetops* primitives manipulate sets of signals. They operate on data objects addressable by the application, not on any set of signals known to the system, such as the set blocked from delivery to a process or the set pending for a process (see 3.3.1.1).

The *sigemptyset*() function initializes the signal set pointed to by the argument set, such that all signals defined in this part of ISO/IEC 9945 are excluded.

The *sigfillset*() function initializes the signal set pointed to by the argument set, such that all signals defined in this part of ISO/IEC 9945 are included.

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 the *pthread_sigmask*(), *sigaction*(), *sigaddset*(), *sigdelset*(), *sigismember*(), *sigpending*(), *sigprocmask*(), *sigsuspend*(), *sigtimedwait*(), *sigwait*(), or *sigwaitinfo*() functions, the results are undefined.

The *sigaddset*() and *sigdelset*() functions respectively add or delete the individual signal specified by the value of the argument *signo* to or from the signal set pointed to by the argument *set*.

The *sigismember*() function tests whether the signal specified by the value of the argument *signo* is a member of the set pointed to by the argument *set*.

3.3.3.3 Returns

Upon successful completion, the *sigismember*() function returns a value of one if the specified signal is a member of the specified set, or a value of zero if it is not. Upon successful completion, the other functions return a value of zero. For all of the above functions, if an error is detected, a value of –1 is returned, and *errno* is set to indicate the error.

3.3.3.4 Errors

For each of the following conditions, if the condition is detected, the *sigaddset*(), *sigdelset*(), and *sigismember*() functions shall return –1 and set *errno* to the corresponding value:

[EINVAL]	The value of the *signo* argument is an invalid or unsupported signal number.

3.3.3.5 Cross-References

sigaction(), 3.3.4; `<signal.h>`, 3.3.1.1; *sigpending*(), 3.3.6; *sigprocmask*(), 3.3.5; *sigsuspend*(), 3.3.7.

3.3.4 Examine and Change Signal Action

Function: *sigaction*()

3.3.4.1 Synopsis

```
#include <signal.h>
int sigaction(int sig, const struct sigaction *act,
            struct sigaction *oact);
```

3.3.4.2 Description

The *sigaction*() function allows the calling process to examine or specify (or both) the action to be associated with a specific signal. The argument *sig* specifies the signal; acceptable values are defined in 3.3.1.1.

The structure *sigaction*, used to describe an action to be taken, is defined in the header `<signal.h>` to include at least the following members:

Member Type	Member Name	Description
void (*)()	*sa_handler*	SIG_DFL, SIG_IGN, or pointer to a function.
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 function.

The storage occupied by *sa_handler* and *sa_sigaction* may overlap, and a conforming application shall not use both simultaneously.

NOTE — Application writers are cautioned that although the current declaration required for *sa_handler* is not a prototype (and thus does not do argument type checking), a future revision of this standard may require a complete prototype. Signal handlers stored in *sa_handler* should be declared to match the single *int* argument implied prototype.

Implementations may add extensions as permitted in 1.3.1.1, item (2). Adding extensions to this structure, which might change the behavior of the application with respect to this standard when those fields in the structure are uninitialized, also requires that the extensions be enabled as required by 1.3.1.1.

If the argument *act*is not **NULL**, it points to a structure specifying the action to be associated with the specified signal. If the argument *oact* is not **NULL**, the action previously associated with the signal is stored in the location pointed to by the argument *oact*. If the argument *act* is **NULL**, signal handling is unchanged by this function call; thus, the call can be used to enquire about the current handling of a given signal. 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 implementation supports the Realtime Signals Extension option and the SA_SIGINFO flag is set in the *sa_flags* field, the *sa_sigaction* field specifies a signal-catching function. If the SA_SIGINFO bit is cleared and the *sa_handler* field specifies a signal-catching function, or if the SA_SIGINFO bit is set, the *sa_mask* field identifies a set of signals that shall be added to the signal mask of the thread before the signal-catching function is invoked. 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.

The *sa_flags* field can be used to modify the behavior of the specified signal.

The following flag bits, defined in the header `<signal.h>`, can be set in *sa_flags*:

Symbolic Constant	Description
SA_NOCLDSTOP	Do not generate SIGCHLD when children stop.
SA_SIGINFO	Invoke the signal-catching function with three arguments instead of one.

If *sig* is SIGCHLD and the SA_NOCLDSTOP flag is not set in *sa_flags*, and the implementation supports the SIGCHLD signal, a SIGCHLD signal shall be generated for the calling process whenever any of its child processes stop. If *sig* is SIGCHLD and the SA_NOCLDSTOP flag is set in *sa_flags*, the implementation shall not generate a SIGCHLD signal in this way.

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 the implementation supports the Realtime Signals Extension option and the SA_SIGINFO flag 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; if delivered, the signal-catching function shall be invoked with three arguments. The application specified value shall be passed to the signal-catching function as the *si_value* member of the *siginfo_t* structure.

When a signal is caught by a signal-catching function installed by the *sigaction*() function, a new signal mask is calculated and installed for the duration of the signal-catching function [or until a call to either the *sigprocmask*() or *sig-suspend*() function 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 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 remains installed until another action is explicitly requested [by another call to the *sigaction*() function] or until one of the *exec* functions is called.

If the previous action for *sig* had been established by the *signal*() function, defined in the C Standard {2}, the values of the fields returned in the structure pointed to by *oact* are unspecified and, in particular, *oact->sv_handler* is not necessarily the same value passed to the *signal*() function. However, if a pointer to the same structure or a copy thereof is passed to a subsequent call to the *sigaction*() function via the act argument, handling of the signal shall be as if the original call to the *signal*() function were repeated.

If the *sigaction*() function 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]. The result of the use of *sigaction*() and a *sigwait* function concurrently within a process on the same signal is unspecified.

3.3.4.3 Returns

Upon successful completion, a value of zero is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

3.3.4.4 Errors

If any of the following conditions occur, the *sigaction*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL]	The value of the *sig* argument is an invalid or unsupported signal number, or an attempt was made to catch a signal that cannot be caught or to ignore a signal that cannot be ignored. See 3.3.1.1.
[ENOTSUP]	The SA_SIGINFO bit flag is set in the *sa_flags* field of the *sigaction* structure, and the implementation does not support the Realtime Signals Extension option.

For each of the following conditions, when the condition is detected and the implementation treats it as an error, the *sigaction*() function shall return a value of −1 and set *errno* to the corresponding value.

[EINVAL]	An attempt was made to set the action to SIG_DFL for a signal that cannot be caught or ignored (or both).

3.3.4.5 Cross-References

kill(), 3.3.2; `<signal.h>`, 3.3.1.1; *sigprocmask*(), 3.3.5; *sigsetops*, 3.3.3.2; *sigsuspend*(), 3.3.7.

3.3.5 Examine and Change Blocked Signals

Functions: *pthread_sigmask*(), *sigprocmask*()

3.3.5.1 Synopsis

```
#include <signal.h>
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oset);
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

3.3.5.2 Description

In a single-threaded process, the *sigprocmask*() function is used to examine or change (or both) the signal mask of the calling thread. The use of the *sigprocmask*() function is unspecified in a multithreaded process. If the value of the argument *set* is not **NULL**, it points to a set of signals to be used to change the currently blocked set.

The value of the argument *how* indicates the manner in which the set is changed and shall consist of one of the following values, as defined in the header `<signal.h>`:

Name	Description
SIG_BLOCK	The resulting set shall be the union of the current set and the signal set pointed to by the argument *set*.
SIG_UNBLOCK	The resulting set shall be the intersection of the current set and the complement of the signal set pointed to by the argument *set*.
SIG_SETMASK	The resulting set shall be the signal set pointed to by the argument *set*.

If the argument *oset* is not **NULL**, the previous mask is stored in the space pointed to by *oset*. If the value of the argument *set* is **NULL**, the value of the argument *how* is not significant and the signal mask of the thread is unchanged by this function call; thus, the call can be used to enquire about currently blocked signals.

If there are any pending unblocked signals after the call to the *sigprocmask*() function, at least one of those signals shall be delivered before the *sigprocmask*() function returns.

It is not possible to block the SIGKILL and SIGSTOP signals; 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 as defined by the C Standard {2}.

If the *sigprocmask*() function fails, the signal mask of the thread is not changed by this function call.

If {_POSIX_THREADS} is defined:

> The *pthread_sigmask*() function is used to examine or change (or both) the signal mask of the calling thread, regardless of the number of threads in the process. The effect shall be the same as described for *sigprocmask*(), without the restriction that the call needs to be made in a single-threaded process.

Otherwise:

> Either the implementation shall support the *pthread_sigmask*() function as described above or the *pthread_sigmask*() function shall not be provided.

3.3.5.3 Returns

Upon successful completion, the *sigprocmask*() function shall return a value of zero. Otherwise, the function shall return a value of −1 and set *errno* to indicate the error. Upon successful completion, the *pthread_sigmask*() function shall return a value of zero. Otherwise, the function shall return an error number.

3.3.5.4 Errors

If any of the following conditions occur, the *sigprocmask*() function shall return −1 ανδ σετ *errno* to the corresponding value:

[EINVAL] The value of the *how* argument is not equal to one of the defined values.

If any of the following conditions occur, the *pthread_sigmask*() function shall return the corresponding error number:

[EINVAL] The value of the *how* argument is not equal to one of the defined values.

3.3.5.5 Cross-References

sigaction(), 3.3.4; `<signal.h>`, 3.3.1.1; *sigpending*(), 3.3.6; *sigsetops*, 3.3.3.2; *sigsuspend*(), 3.3.7.

3.3.6 Examine Pending Signals

Function: *sigpending*()

3.3.6.1 Synopsis

```
#include <signal.h>
int sigpending(sigset_t *set);
```

3.3.6.2 Description

The *sigpending*() function shall store, in the location referenced by the *set* argument, the set of signals that are blocked from delivery and are pending either for the process or the calling thread.

3.3.6.3 Returns

Upon successful completion, a value of zero is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

3.3.6.4 Errors

This part of ISO/IEC 9945 does not specify any error conditions that are required to be detected for the *sigpending*() function. Some errors may be detected under conditions that are unspecified by this part of ISO/IEC 9945.

3.3.6.5 Cross-References

`<signal.h>`, 3.3.1.1; *sigprocmask*(), 3.3.5; *sigsetops*, 3.3.3.2.

3.3.7 Wait for a Signal

Function: *sigsuspend*()

3.3.7.1 Synopsis

```
#include <signal.h>
int sigsuspend(const sigset_t *sigmask);
```

3.3.7.2 Description

The *sigsuspend*() function replaces the signal mask of the thread with the set of signals pointed to by the argument *sigmask* and then suspends the calling thread until delivery of a signal to the calling thread 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 o the thread.

If the action is to terminate the process, the *sigsuspend*() function shall not return. If the action is to execute a signal-catching function, the *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 those signals that cannot be ignored, as documented in 3.3.1.1; this shall be enforced by the system without causing an error to be indicated.

3.3.7.3 Returns

Since the *sigsuspend*() function suspends process execution indefinitely, there is no successful completion return value. A value of –1 is returned and *errno* is set to indicate the error.

3.3.7.4 Errors

If any of the following conditions occur, the *sigsuspend*() function shall return –1 and set *errno* to the corresponding value:

[EINTR]	A signal is caught by the calling process, and control is returned from the signal-catching function.

3.3.7.5 Cross-References

pause(), 3.4.2; *sigaction*(), 3.3.4; `<signal.h>`, 3.3.1.1; *sigpending*(), 3.3.6; *sigprocmask*(), 3.3.5; *sigsetops*, 3.3.3.2.

3.3.8 Synchronously Accept a Signal

Function: *sigwait*(), *sigwaitinfo*(), *sigtimedwait*()

3.3.8.1 Synopsis

```
#include <signal.h>
int sigwait(const sigset_t *set, int *sig);
int sigwaitinfo(const sigset_t *set, siginfo_t *info);
int sigtimedwait(const sigset_t *set, siginfo_t *info,
            const struct timespec *timeout);
```

3.3.8.2 Description

This subclause defines the family of *sigwait* functions.

The *sigwait*() function selects a pending signal from *set*, atomically clears it from the set of pending signals in the system, and returns that signal number in the location referenced by *sig*. If prior to the call to *sigwait*() there are multiple pending instances of a single signal number, it is implementation defined whether upon successful return there are any remaining pending signals for that signal number. If the implementation supports queued signals and there are multiple signals queued for the signal number selected, the first such queued signal shall cause a return from *sigwait*() and the remainder shall remain queued. If no signal in *set* is pending at the time of the call, the thread shall be suspended until one or more becomes pending. The signals defined by set shall have been blocked at the time of the call to *sigwait*(); otherwise, the behavior is undefined. The effect of *sigwait*() on the signal actions for the signals in *set* is unspecified.

If more than one thread is using *sigwait*() to wait for the same signal, no more than one of these threads shall return from *sigwait*() with the signal number. Which thread returns from *sigwait*() if more than a single thread is waiting is unspecified.

If {_POSIX_REALTIME_SIGNALS} is defined:

> 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 nonrealtime signals, or between multiple pending nonrealtime signals, is unspecified.
> The function *sigwaitinfo*() behaves the same as the *sigwait*() function if the *info* argument is **NULL**. If the *info* argument is non-**NULL**, the *sigwaitinfo*() function behaves the same as *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 made available to queue other signals. 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.
> The function *sigtimedwait*() behaves the same as *sigwaitinfo*() except that if none of the signals specified by *set* are pending, *sigtimedwait*() shall wait for the time interval specified in the *timespec* structure referenced by *timeout*. If the *timespec* structure pointed to by *timeout* is zero-valued and if none of the signals specified by *set* are pending, then *sigtimedwait*() shall return immediately with an error. If *timeout* is the **NULL** pointer, the behavior is unspecified.

Otherwise:

> Either the implementation shall support the *sigwaitinfo*() and *sigtimedwait*() functions as described above or each of the *sigwaitinfo*()and *sigtimedwait*() functions shall fail.

3.3.8.3 Returns

Upon successful completion (that is, one of the signals specified by *set* is pending or has been generated) *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. Upon successful completion *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.

3.3.8.4 Errors

For each of the following conditions, if the condition is detected, the *sigwait*() function shall return the corresponding error number:

[EINVAL] The *set* argument contains an invalid or unsupported signal number.

If any of the following conditions occur, the *sigwaitinfo*() and *sigtimedwait*() functions shall return –1 and set *errno* to the corresponding value:

[ENOSYS] The functions *sigwaitinfo*() and *sigtimedwait*() are not supported by this implementation.

For each of the following conditions, if the condition is detected, the *sigwaitinfo*() and *sigtimedwait*() functions shall return –1 and set *errno* to the corresponding value:

[EINTR] The wait was interrupted by an unblocked, caught signal. It shall be documented in the system documentation whether this error is returned.

If any of the following conditions occur, the *sigtimedwait*() function shall return –1 and set *errno* to the corresponding value:

[EAGAIN] No signal specified by set was generated within the specified timeout period.

For each of the following conditions, if the condition is detected, the *sigtimedwait*() function shall return –1 and set *errno* to the corresponding value:

[EINVAL] The *timeout* argument specified a *tv_nsec* value less than zero or greater than or equal to 1000 million.

An implementation should only check for this error if no signal is pending in *set* and it is necessary to wait.

3.3.8.5 Cross-References

pause(), 3.4.2; *pthread_sigmask*(), 3.3.5; *sigaction*(), 3.3.4; *sigpending*(), 3.3.6; *sigsuspend*(), 3.3.7; `<signal.h>`, 3.3.1.1; `<time.h>`, 14.1.

3.3.9 Queue a Signal to a Process

Function: *sigqueue*()

3.3.9.1 Synopsis

```
#include <signal.h>
int sigqueue(pid_t pid, int signo, const union sigval value);
```

3.3.9.2 Description

If {_POSIX_REALTIME_SIGNALS} is defined:

The *sigqueue*() function causes 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 are 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 one pending unblocked signal shall be delivered to the calling thread before the *sigqueue*() function returns. Should any of 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 nonrealtime signals, or between multiple pending nonrealtime signals, is unspecified.

Otherwise:

Either the implementation shall support the *sigqueue*() function as described above or the *sigqueue*() function shall fail.

3.3.9.3 Returns

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.

3.3.9.4 Errors

If any of the following conditions occur, the *sigqueue*() function shall return −1 and set *errno* to the corresponding value:

[EAGAIN]	No resources available to queue the signal. The process has already queued {SIGQUEUE_MAX} signals that are still pending at the receiver(s), or a systemwide resource limit has been exceeded.
[EINVAL]	The value of the *signo* argument is an invalid or unsupported signal number.
[ENOSYS]	The function *sigqueue*() is not supported by this implementation.
[EPERM]	The process does not have the appropriate privilege to send the signal to the receiving process.
[ESRCH]	The process *pid* does not exist.

3.3.9.5 Cross-References

`<signal.h>`, 3.3.1.1.

3.3.10 Send a Signal to a Thread

Function: *pthread_kill*()

3.3.10.1 Synopsis

```
#include <signal.h>
int pthread_kill(pthread_t thread, int sig);
```

3.3.10.2 Description

If {POSIX_THREADS} is defined:

The *pthread_kill*() function is used to request that a signal be delivered to the specified thread.
As in *kill*(), if *sig* is zero, error checking is performed but no signal is actually sent.

Otherwise:

Either the implementation shall support the *pthread_kill*() function as described above or the *pthread_kill*() function shall not be provided.

3.3.10.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, the function shall return an error number. If the *pthread_kill*() function fails, no signal shall be sent.

3.3.10.4 Errors

If any of the following conditions occur, the *pthread_kill*() function shall return the corresponding error number:

[ESRCH]	No thread could be found corresponding to that specified by the given thread ID.
[EINVAL]	The value of the *sig* argument is an invalid or unsupported signal number.

3.3.10.5 Cross-References

kill(), 3.3.2; *pthread_self*(), 16.2.6; *raise*(), 8.1.

3.4 Timer Operations

A thread can suspend itself for a specific period of time with the *sleep*() function or suspend itself indefinitely with the *pause*() function until a signal is delivered to the thread. The *alarm*() function schedules a signal to be generated for the process at a specific time.

3.4.1 Schedule Alarm

Function: *alarm*()

3.4.1.1 Synopsis

```
unsigned int alarm(unsigned int seconds);
```

3.4.1.2 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 cause the process actually not to begin handling the signal until after the desired time.

Alarm requests are not stacked; only one SIGALRM generation can be scheduled in this manner. If the SIGALRM has not yet been generated, the call will result in rescheduling the time at which the SIGALRM will be generated.

If *seconds* is zero, any previously made *alarm*() request is canceled.

3.4.1.3 Returns

If there is a previous *alarm*() request with time remaining, the *alarm*() function shall return a nonzero value that is the number of seconds until the previous request would have generated a SIGALRM signal. Otherwise, the *alarm*() function shall return zero.

3.4.1.4 Errors

The *alarm*() function is always successful, and no return value is reserved to indicate an error.

3.4.1.5 Cross-References

exec, 3.1.2; *fork*(), 3.1.1; *pause*(), 3.4.2; *sigaction*(), 3.3.4; `<signal.h>`, 3.3.1.1.

3.4.2 Suspend Process Execution

Function: *pause*()

3.4.2.1 Synopsis

```
int pause(void);
```

3.4.2.2 Description

The *pause*() function suspends the calling thread until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process.

If the action is to terminate the process, the *pause*() function shall not return.

If the action is to execute a signal-catching function, the *pause*() function shall return after the signal-catching function returns.

3.4.2.3 Returns

Since the *pause*() function suspends thread execution indefinitely, there is no successful completion return value. A value of –1 is returned and *errno* is set to indicate the error.

3.4.2.4 Errors

If any of the following conditions occur, the *pause*() function shall return –1 ανδ σετ *errno* to the corresponding value:

[EINTR] A signal is caught by the calling process, and control is returned from the signal-catching function.

3.4.2.5 Cross-References

alarm(), 3.4.1; *kill*(), 3.3.2; *wait*, 3.2.1; 3.3.1.4.

3.4.3 Delay Process Execution

Function: *sleep*()

3.4.3.1 Synopsis

```
unsigned int sleep(unsigned int seconds);
```

3.4.3.2 Description

The *sleep*() function shall cause the current thread to be suspended from execution until either the number of realtime seconds specified by the argument *seconds* have elapsed or a signal is delivered to the calling thread and its action is to invoke a signal-catching function or to terminate the process. The suspension time may be longer than requested due to the scheduling of other activity by the system.

If a SIGALRM signal is generated for the calling process during execution of the *sleep*() function and the SIGALRM signal is being ignored or blocked from delivery, it is unspecified whether *sleep*() returns when the SIGALRM signal is scheduled. If the signal is being blocked, it is also unspecified whether it remains pending after the *sleep*() function returns or is discarded.

If a SIGALRM signal is generated for the calling process during execution of the *sleep*() function, except as a result of a prior call to the *alarm*() function, and if the SIGALRM signal is not being ignored or blocked from delivery, it is unspecified whether that signal has any effect other than causing the *sleep*() function to return.

If a signal-catching function interrupts the *sleep*() function and either examines or changes the time a SIGALRM is scheduled to be generated, the action associated with the SIGALRM signal, or whether the SIGALRM signal is blocked from delivery, the results are unspecified.

If a signal-catching function interrupts the *sleep*() function and calls the *siglongjmp*() or *longjmp*() function to restore an environment saved prior to the *sleep*() call, the action associated with the SIGALRM signal and the time at which a SIGALRM signal is scheduled to be generated are unspecified. It is also unspecified whether the SIGALRM signal is blocked, unless the process's signal mask is restored as part of the environment (see 8.3.1).

3.4.3.3 Returns

If the *sleep*() function returns because the requested time has elapsed, the value returned shall be zero. If the *sleep*() function returns due to delivery of a signal, the value returned shall be the unslept amount (the requested time minus the time actually slept) in seconds.

3.4.3.4 Errors

The *sleep*() function is always successful, and no return value is reserved to indicate an error.

3.4.3.5 Cross-References

alarm(), 3.4.1; *pause*(), 3.4.2; *sigaction*(), 3.3.4.

4. Process Environment

4.1 Process Identification

4.1.1 Get Process and Parent Process IDs

Functions: *getpid*(), *getppid*()

4.1.1.1 Synopsis

```
#include <sys/types.h>
pid_t getpid(void);
pid_t getppid(void);
```

4.1.1.2 Description

The *getpid*() function returns the process ID of the calling process.

The *getppid*() function returns the parent process ID of the calling process.

4.1.1.3 Returns

See 4.1.1.2.

4.1.1.4 Errors

The *getpid*() and *getppid*() functions are always successful, and no return value is reserved to indicate an error.

4.1.1.5 Cross-References

exec, 3.1.2; *fork*(), 3.1.1; *kill*(), 3.3.2.

4.2 User Identification

4.2.1 Get Real User, Effective User, Real Group, and Effective Group IDs

Functions: *getuid*(), *geteuid*(), *getgid*(), *getegid*()

4.2.1.1 Synopsis

```
#include <sys/types.h>
uid_t getuid(void);
uid_t geteuid(void);
gid_t getgid(void);
gid_t getegid(void);
```

4.2.1.2 Description

The *getuid*() function returns the real user ID of the calling process.

The *geteuid*() function returns the effective user ID of the calling process.

The *getgid*() function returns the real group ID of the calling process.

The *getegid*() function returns the effective group ID of the calling process.

4.2.1.3 Returns

See 4.2.1.2.

4.2.1.4 Errors

The *getuid*(), *geteuid*(), *getgid*(), and *getegid*() functions are always successful, and no return value is reserved to indicate an error.

4.2.1.5 Cross-References

setuid(), 4.2.2.

4.2.2 Set User and Group IDs

Functions: *setuid*(), *setgid*()

4.2.2.1 Synopsis

```
#include <sys/types.h>
int setuid(uid_t uid);
int setgid(gid_t gid);
```

4.2.2.2 Description

If {_POSIX_SAVED_IDS} is defined:

1) If the process has appropriate privileges, the *setuid*() function sets the real user ID, effective user ID, and the saved set-user-ID to *uid*.
2) If the process does not have appropriate privileges, but *uid* is equal to the real user ID or the saved set-user-ID, the *setuid*() function sets the effective user ID to *uid*; the real user ID and saved set-user-ID remain unchanged by this function call.
3) If the process has appropriate privileges, the *setgid*() function sets the real group ID, effective group ID, and the saved set-group-ID to *gid*.
4) If the process does not have appropriate privileges, but *gid* is equal to the real group ID or the saved set-group-ID, the *setgid*() function sets the effective group ID to *gid*; the real group ID and saved set-group-ID remain unchanged by this function call.

Otherwise:

1) If the process has appropriate privileges, the *setuid*() function sets the real user ID and effective user ID to *uid*.
2) If the process does not have appropriate privileges, but *uid* is equal to the real user ID, the *setuid*() function sets the effective user ID to *uid*; the real user ID remains unchanged by this function call.
3) If the process has appropriate privileges, the *setgid*() function sets the real group ID and effective group ID to *gid*.
4) If the process does not have appropriate privileges, but *gid* is equal to the real group ID, the *setgid*() function sets the effective group ID to *gid*; the real group ID remains unchanged by this function call.

Any supplementary group IDs of the calling process remain unchanged by these function calls.

4.2.2.3 Returns

Upon successful completion, a value of zero is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

4.2.2.4 Errors

If any of the following conditions occur, the *setuid*() function shall return −1 and set *errno* to the corresponding value:

[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, if {_POSIX_SAVED_IDS} is defined, the saved set-user-ID.

If any of the following conditions occur, the *setgid*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL]	The value of the *gid* argument is invalid and not supported by the implementation.
[EPERM]	The process does not have appropriate privileges and *gid* does not match the real group ID or, if {_POSIX_SAVED_IDS} is defined, the saved set-group-ID.

4.2.2.5 Cross-References

exec, 3.1.2; *getuid*(), 4.2.1.

4.2.3 Get Supplementary Group IDs

Function: *getgroups*()

4.2.3.1 Synopsis

```
#include <sys/types.h>
int getgroups(int gidsetsize, gid_t grouplist[]);
```

4.2.3.2 Description

The *getgroups*() function fills in the array *grouplist* with the supplementary group IDs of the calling process. The *gidsetsize* argument specifies the number of elements in the supplied array *grouplist*. The actual number of supplementary group IDs stored in the array is returned. The values of array entries with indices larger than or equal to the returned value are undefined.

As a special case, if the *gidsetsize* argument is zero, *getgroups*() returns the number of supplementary group IDs associated with the calling process without modifying the array pointed to by the *grouplist* argument.

4.2.3.3 Returns

Upon successful completion, the number of supplementary group IDs is returned. This value is zero if {NGROUPS_MAX} is zero. A return value of –1 indicates 109 failure, and *errno* is set to indicate the error.

4.2.3.4 Errors

If any of the following conditions occur, the *getgroups*() function shall return –1 ανδ σετ *errno* to the corresponding value:

[EINVAL]	The *gidsetsize* argument is not equal to zero and is less than the number of supplementary group IDs.

4.2.3.5 Cross-References

setgid(), 4.2.2.

4.2.4 Get User Name

Functions: *getlogin*(), *getlogin_r*() |

4.2.4.1 Synopsis

```
char *getlogin(void);
int getlogin_r(char *name, size_t namesize);                    |
```

4.2.4.2 Description

The *getlogin*() function returns a pointer to a string giving a user name associated with the calling process, which is the login name associated with the calling process.

If *getlogin*() returns a non-**NULL** pointer, that pointer points to the name under which the user logged in, even if there are several login names with the same user ID.

If {_POSIX_THREAD_SAFE_FUNCTIONS} is defined:

> The *getlogin_r*() function puts the name associated by the login activity with the control terminal of the current process in the character array pointed to by *name*. The array is *namesize* characters long and should have space for the name and the terminating null character. The maximum size of the login name is {LOGIN_NAME_MAX}.
> If *getlogin_r*() is successful, *name* points to the name the user used at login, even if there are several login names with the same user ID.

Otherwise:

> Either the implementation shall support the *getlogin_r*() function as described above or the *getlogin_r*() function shall not be provided.

4.2.4.3 Returns

The *getlogin*() function returns a pointer to a string containing the user's login name, or a **NULL** pointer if the user's login name cannot be found.

The return value from *getlogin*() may point to static data and, therefore, may be overwritten by each call.

If successful, the *getlogin_r*() function shall return zero. Otherwise, an error number shall be returned to indicate the error.

4.2.4.4 Errors

This part of ISO/IEC 9945 does not specify any error conditions that are required to be detected for the *getlogin*() function. Some errors may be detected under conditions that are unspecified by this part of ISO/IEC 9945.

For each of the following conditions, if the condition is detected, the *getlogin_r*() function shall return the corresponding error number:

[ERANGE]	The value of *namesize* is smaller than the length of the string to be returned, including the terminating null character.

4.2.4.5 Cross-References

getpwnam(), 9.2.2; *getpwuid*(), 9.2.2.

4.3 Process Groups

4.3.1 Get Process Group ID

Function: *getpgrp*()

4.3.1.1 Synopsis

```
#include <sys/types.h>
pid_t getpgrp(void);
```

4.3.1.2 Description

The *getpgrp*() function returns the process group ID of the calling process.

4.3.1.3 Returns

See 4.3.1.2.

4.3.1.4 Errors

The *getpgrp*() function is always successful, and no return value is reserved to indicate an error.

4.3.1.5 Cross-References

setpgid(), 4.3.3; *setsid*(), 4.3.2; *sigaction*(), 3.3.4.

4.3.2 Create Session and Set Process Group ID

Function: *setsid*()

4.3.2.1 Synopsis

```
#include <sys/types.h>
pid_t setsid(void);
```

4.3.2.2 Description

If the calling process is not a process group leader, the *setsid*() function shall create a new session. 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.

4.3.2.3 Returns

Upon successful completion, the *setsid*() function returns the value of the process group ID of the calling process. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

4.3.2.4 Errors

If any of the following conditions occur, the *setsid*() function shall return –1 and set *errno* to the corresponding value:

[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.

4.3.2.5 Cross-References

exec, 3.1.2; *_exit*(), 3.2.2; *fork*(), 3.1.1; *getpid*(), 4.1.1; *kill*(), 3.3.2; *setpgid*(), 4.3.3; *sigaction*(), 3.3.4.

4.3.3 Set Process Group ID for Job Control

Function: *setpgid*()

4.3.3.1 Synopsis

```
#include <sys/types.h>
int setpgid(pid_t pid, pid_t pgid);
```

4.3.3.2 Description

If {_POSIX_JOB_CONTROL} is defined:

> The *setpgid*() function is used to 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 zero, the process ID of the calling process shall be used. Also, if *pgid* is zero, the process ID of the indicated process shall be used.

Otherwise:

> Either the implementation shall support the *setpgid*() function as described above or the *setpgid*() function shall fail.

4.3.3.3 Returns

Upon successful completion, the *setpgid*() function returns a value of zero. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

4.3.3.4 Errors

If any of the following conditions occur, the *setpgid*() function shall return –1 and set *errno* to the corresponding value:

[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 zero or is not a value supported by the implementation.
[ENOSYS]	The *setpgid*() function is not supported by this implementation.
[EPERM]	The process indicated by the *pid* argument is a session leader.
	The value of the *pid* argument is valid, but 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.
	The value of the *pgid* argument 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.

4.3.3.5 Cross-References

getpgrp(), 4.3.1; *setsid*(), 4.3.2; *tcsetpgrp*(), 7.2.4; *exec*, 3.1.2.

4.4 System Identification

4.4.1 Get System Name

Function: *uname*()

4.4.1.1 Synopsis

```
#include <sys/utsname.h>
int uname(struct utsname *name);
```

4.4.1.2 Description

The *uname*() function stores information identifying the current operating system in the structure pointed to by the argument *name*.

The structure *utsname* is defined in the header `<sys/utsname.h>` and contains at least the members shown in Table 4.1.

Table 4.1—*uname*() Structure Members

Member Name	Description
sysname	Name of this implementation of the operating system.
nodename	Name of this node within an implementation-specified communications network.
release	Current release level of this implementation.
version	Current version level of this release.
machine	Name of the hardware type on which the system is running.

Each of these data items is a null-terminated array of *char*.

The format of each member is implementation defined. The system documentation (see 1.3.1.2) shall specify the source and format of each member and may 261 specify the range of values for each member.

The inclusion of the *nodename*, member in this structure does not imply that it is sufficient information for interfacing to communications networks.

4.4.1.3 Returns

Upon successful completion, a nonnegative value is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

4.4.1.4 Errors

This part of ISO/IEC 9945 does not specify any error conditions that are required to be detected for the *uname*() function. Some errors may be detected under conditions that are unspecified by this part of ISO/IEC 9945.

4.5 Time

4.5.1 Get System Time

Function: *time*()

4.5.1.1 Synopsis

```
#include <time.h>
time_t time(time_t *tloc);
```

4.5.1.2 Description

The *time*() function returns the value of time in seconds since the Epoch.

The argument *tloc* points to an area where the return value is also stored. If *tloc* is a **NULL** pointer, no value is stored.

4.5.1.3 Returns

Upon successful completion, *time*() returns the value of time. Otherwise, a value of ((*time_t*) −1) is returned and *errno* is set to indicate the error.

4.5.1.4 Errors

This part of ISO/IEC 9945 does not specify any error conditions that are required to be detected for the *time*() function. Some errors may be detected under conditions that are unspecified by this part of ISO/IEC 9945.

4.5.2 Get Process Times

Function: *times*()

4.5.2.1 Synopsis

```
#include <sys/times.h>
clock_t times(struct tms *buffer);
```

4.5.2.2 Description

The *times*() function shall fill the structure pointed to by *buffer* with time accounting information. The type *clock_t* and the *tms* structure are defined in `<sys/times.h>`; the *tms* structure shall contain at least the following members:

Member Type	**Member Name**	**Description**
clock_t	*tms_utime*	User CPU time.
clock_t	*tms_stime*	System CPU time.
clock_t	*tms_cutime*	User CPU time of terminated child processes.
clock_t	*tms_cstime*	System CPU time of terminated child processes.

All times are measured in terms of the number of clock ticks used.

The times of a terminated child process are included in the *tms_cutime* and *tms_cstime* elements of the parent when a *wait*() or *waitpid*() function returns the 306 process ID of this terminated child. See 3.2.1. If a child process has not waited 307 for its terminated children, their times shall not be included in its times.

The value *tms_utime* is the CPU time charged for the execution of user instructions.

The value *tms_stime* is the CPU time charged for execution by the system on behalf of the process.

The value *tms_cutime* is the sum of the *tms_utimes* and *tms_cutimes* of the child processes.

The value *tms_cstime* is the sum of the *tms_stimes* and *tms_cstimes* of the child processes.

4.5.2.3 Returns

Upon successful completion, *times*() shall return the elapsed real time, in clock ticks, since an arbitrary point in the past (for example, system start-up time). This point does not change from one invocation of *times*() within the process to another. The return value may overflow the possible range of type *clock_t*. If the *times*() function fails, a value of ((*clock_t*) −1) is returned and *errno* is set to indicate the error.

4.5.2.4 Errors

This part of ISO/IEC 9945 does not specify any error conditions that are required to be detected for the *times*() function. Some errors may be detected under conditions that are unspecified by this part of ISO/IEC 9945.

4.5.2.5 Cross-References

exec, 3.1.2; *fork*(), 3.1.1; *sysconf*(), 4.8.1; *time*(), 4.5.1; *wait*(), 3.2.1.

4.6 Environment Variables

4.6.1 Environment Access

Function: *getenv*

4.6.1.1 Synopsis

```
#include <stdlib.h>
char *getenv(const char *name);
```

4.6.1.2 Description

The *getenv*() function searches the environment list (see 2.6) for a string of the form *name*=*value* and returns a pointer to *value* if such a string is present. If the specified *name* cannot be found, a **NULL** pointer is returned.

4.6.1.3 Returns

Upon successful completion, the *getenv*() function returns a pointer to a string containing the *value* for the specified *name*, or a **NULL** pointer if the specified *name* cannot be found. The return value from *getenv*() may point to static data and, therefore, may be overwritten by each call. Unsuccessful completion shall result in the return of a **NULL** pointer.

4.6.1.4 Errors

This part of ISO/IEC 9945 does not specify any error conditions that are required to be detected for the *getenv*() function. Some errors may be detected under conditions that are unspecified by this part of ISO/IEC 9945.

4.6.1.5 Cross-References

3.1.2; 2.6.

4.7 Terminal Identification

4.7.1 Generate Terminal Pathname

Function: *ctermid*()

4.7.1.1 Synopsis

```
#include <stdio.h>
char *ctermid(char *s);
```

4.7.1.2 Description

The *ctermid*() function generates a string that, when used as a pathname, refers to the current controlling terminal for the current process.

If the *ctermid*() function returns a pathname, access to the file is not guaranteed. If the application uses any of the interfaces guaranteed to be available if either {_POSIX_THREAD_SAFE_FUNCTIONS} or {_POSIX_THREADS} is defined, the *ctermid*() function shall be called with a non-**NULL** parameter.

4.7.1.3 Returns

If *s* is a **NULL** pointer, the string is generated in an area that may be static (and, therefore, may be overwritten by each call), the address of which is returned. Otherwise, *s* is assumed to point to an array of *char* of at least L_ctermid bytes; the string is placed in this array and the value of *s* is returned. The symbolic constant L_ctermid is defined in `<stdio.h>` and shall have a value greater than zero.

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.

4.7.1.4 Errors

This part of ISO/IEC 9945 does not specify any error conditions that are required to be detected for the *ctermid*() function. Some errors may be detected under conditions that are unspecified by this part of ISO/IEC 9945.

4.7.1.5 Cross-References

ttyname(), 4.7.2.

4.7.2 Determine Terminal Device Name

Functions: *ttyname*(), *ttyname_r*(), *isatty*()

4.7.2.1 Synopsis

```
char *ttyname(int fildes);
int ttyname_r(int fildes, char *name, size_t namesize);
int isatty(int fildes);
```

4.7.2.2 Description

The *ttyname*() function returns a pointer to a string containing a null-terminated pathname of the terminal associated with file descriptor *fildes*.

The return value of *ttyname*() may point to static data that is overwritten by each call.

The *isatty*() function returns 1 if *fildes* is a valid file descriptor associated with a terminal, zero otherwise.

If {_POSIX_THREAD_SAFE_FUNCTIONS} is defined:

> The *ttyname_r*() function stores 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 is {TTY_NAME_MAX}.

Otherwise:

> Either the implementation shall support the *ttyname_r*() function as described above or the *ttyname_r*() function shall not be provided.

4.7.2.3 Returns

The *ttyname*() function returns a **NULL** pointer if *fildes* is not a valid file descriptor associated with a terminal or if the pathname cannot be determined.

If successful, the *ttyname_r*() function shall return zero. Otherwise, an error number shall be returned to indicate the error.

4.7.2.4 Errors

This part of ISO/IEC 9945 does not specify any error conditions that are required to be detected for the *ttyname*() or *isatty*() functions. Some errors may be detected under conditions that are unspecified by this part of ISO/IEC 9945.

For each of the following conditions, if the condition is detected, the *ttyname_r*() function shall return the corresponding error number:

[EBADF]	The *filedes* argument is not a valid file descriptor.
[ENOTTY]	The *filedes* argument does not refer to a tty.
[ERANGE]	The value of *namesize* is smaller than the length of the string to be returned, including the terminating null character.

4.8 Configurable System Variables

4.8.1 Get Configurable System Variables

Function: *sysconf*()

4.8.1.1 Synopsis

```
#include <unistd.h>
long sysconf(int name);
```

4.8.1.2 Description

The *sysconf*() function provides a method for the application to determine the current value of a configurable system limit or option (*variable*).

The *name* argument represents the system variable to be queried. The implementation shall support all of the variables listed in Table 4.2 and may support others. The variables in Table 4.2 come from `<limits.h>` or `<unistd.h>` and the symbolic constants, defined in `<unistd.h>`, that are the corresponding values used for *name*.

Table 4.2—Configurable System Variables

Variable	*name* Value
{AIO_LISTIO_MAX}	{_SC_AIO_LISTIO_MAX}
{AIO_ MAX}	{_SC_AIO_MAX}
{AIO_PRIO_DELTA_MAX}	{SC_AIO_PRIO_DELTA_MAX}
{ARG_MAX}	{_SC_ARG_MAX}
{CHILD_MAX}	{_SC_CHILD_MAX}
clock ticks/second	{_SC_CLK_TCK}
{DELAYTIMER_MAX}	{_SC_DELAYTIMER_MAX}
Maximum size of *getgrgid_r*() and *getgrnam_r*() data buffers	{_SC_GETGR_R_SIZE_MAX}
Maximum size of *getpwuid_r*() and *getpwnam_r*() data buffers	{_SC_GETPW_R_SIZE_MAX}
{LOGIN_NAME_MAX}	{_SC_LOGIN_NAME_MAX}
{MQ_OPEN_MAX}	{_SC_MQ_OPEN_MAX}
{MQ_PRIO_MAX}	{_SC_MQ_PRIO_MAX}
{NGROUPS_MAX}	{_SC_NGROUPS_MAX}
{OPEN_MAX}	{_SC_OPEN_MAX}
{PAGESIZE}	{_SC_PAGESIZE}
{RTSIG_MAX}	{_SC_RTSIG_MAX}
{SEM_NSEMS_MAX}	{_SC_SEM_NSEMS_MAX}
{SEM_VALUE_MAX}	{_SC_SEM_VALUE_MAX}
{SIGQUEUE_MAX}	{_SC_SIGQUEUE_MAX}

Table 4.2—Configurable System Variables (Continued)

Variable	*name* Value
{STREAM_MAX}	{_SC_STREAM_MAX}
{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}
{TIMER_MAX}	{_SC_TIMER_MAX}
{TTY_NAME_MAX}	{_SC_TTY_NAME_MAX}
{TZNAME_MAX}	{_SC_TZNAME_MAX}
{_POSIX_ASYNCHRONOUS_IO}	{_SC_ASYNCHRONOUS_IO}
{_POSIX_FSYNC}	{_SC_FSYNC}
{_POSIX_JOB_CONTROL}	{_SC_JOB_CONTROL}
{_POSIX_MAPPED_FILES}	{_SC_MAPPED_FILES}
{_POSIX_MEMLOCK}	{_SC_MEMLOCK}
{_POSIC_MEMLOCK_RANGE}	{_SC_MEMLOCK_RANGE}
{_POSIX_MEMORY_PROTECTION}	{_SC_MEMORY_PROTECTION}
{_POSIX_MESSAGE_PASSING}	{_SC_MESSAGE_PASSING}
{_POSIX_PRIORITIZED_IO}	{_SC_PRIORITIZED_IO}
{_POSIX_PRIORITY_SCHEDULING}	{_SC_PRIORITY_SCHEDULING}
{_POSIX_REALTIME_SIGNALS}	{_SC_REALTIME_SIGNALS}
{_POSIX_SAVED_IDS}	{_SC_SAVED_IDS}
{_POSIX_SEMAPHORES}	{_SC_SEMAPHORES}
{_POSIX_SHARED_MEMORY_OBJECTS}	{_SC_SHARED_MEMORY_OBJECTS}
{_POSIX_SYNCHRONIZED_IO}	{_SC_SYNCHRONIZED_IO}
{_POSIX_TIMERS}	{_SC_TIMERS}
{_POSIX_THREADS}	{_SC_THREADS}
{_POSIX_THREAD_ATTR_STACKADDR}	{_SC_THREAD_ATTR_STACKADDR}
{_POSIX_THREAD_ATTR_STACKSIZE}	{_SC_THREAD_ATTR_STACKSIZE}
{_POSIX_THREAD_PRIORITY_SCHEDULING}	{_SC_THREAD_PRIORITY_SCHEDULING}
{_POSIX_THREAD_PRIO_INHERIT}	{_SC_THREAD_PRIO_INHERIT}
{_POSIX_THREAD_PRIO_PROTECT}	{_SC_THREAD_PRIO_PROTECT}
{_POSIX_THREAD_PROCESS_SHARED}	{_SC_THREAD_PROCESS_SHARED}
{_POSIX_THREAD_SAFE_FUNCTIONS}	{_SC_THREAD_SAFE_FUNCTIONS}
{_POSIX_VERSION}	{_SC_VERSION}

4.8.1.3 Returns

If *name* is an invalid value, *sysconf*() shall return –1. If the variable corresponding to *name* is associated with functionality that is not supported by the system, *sysconf*() shall return –1 without changing the value of *errno*.

Otherwise, the *sysconf*() function returns the current variable value on the system. The value returned shall not be more restrictive than the corresponding value described to the application when it was compiled with the implementation's `<limits.h>` or `<unistd.h>`. The value shall not change during the lifetime of the calling process.

4.8.1.4 Errors

If any of the following conditions occur, the *sysconf*() function shall return –1 and set *errno* to the corresponding value:

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

4.8.1.5 Special Symbol {CLK_TCK}

The special symbol {CLK_TCK} shall yield the same result as sysconf (_SC_CLK_TCK). It shall be defined in `<time.h>`. The symbol {CLK_TCK} may be evaluated by the implementation at run time or may be a constant. This special symbol is obsolescent.

5. Files and Directories

The functions in this section perform the operating system services dealing with the creation and removal of files and directories and the detection and modification of their characteristics. They also provide the primary methods a process will use to gain access to files and directories for subsequent I/O operations (see Section 6).

5.1 Directories

5.1.1 Format of Directory Entries

The header `<dirent.h>` defines a structure and a defined type used by the *directory* routines.

The internal format of directories is unspecified.

The *readdir*() function returns a pointer to an object of type *struct dirent* that includes the member:

Member Type	Member Name	Description
char []	*d_name*	Null-terminated filename

The array of *char d_name* is of unspecified size, but the number of bytes preceding the terminating null character shall not exceed {NAME_MAX}.

5.1.2 Directory Operations

Functions: *opendir*(), *readdir*(), *readdir_r*(), *rewinddir*(), *closedir*()

5.1.2.1 Synopsis

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *dirname);
struct dirent *readdir(DIR *dirp);
int readdir_r(DIR *dirp, struct dirent *entry, struct dirent **result);
void rewinddir(DIR *dirp);
int closedir(DIR *dirp);
```

5.1.2.2 Description

The type *DIR*, which is defined in the header `<dirent.h>`, represents a *directory stream*, which is an ordered sequence of all the directory entries in a particular directory. Directory entries represent files; files may be removed from a directory or added to a directory asynchronously to the operations described in this subclause (5.1.2). The type *DIR* may be implemented using a file descriptor. In that case, applications will only be able to open up to a total of {OPEN_MAX} files and directories; see 5.3.1. A successful call to any of the *exec* functions shall close any directory streams that are open in the calling process.

The *opendir*() function opens a directory stream corresponding to the directory named by the *dirname* argument. The directory stream is positioned at the first entry.

The *readdir*() function returns a pointer to a structure representing the directory entry at the current position in the directory stream to which *dirp* refers, and positions the directory stream at the next entry. It returns a **NULL** pointer upon reaching the end of the directory stream.

The *readdir*() function shall not return directory entries containing empty names. It is unspecified whether entries are returned for dot or dot-dot.

The pointer returned by *readdir*() points to data that may be overwritten by another call to *readdir*() on the same directory stream. This data shall not be overwritten by another call to *readdir*() on a different directory stream.

The *readdir*() function may buffer several directory entries per actual read operation; the *readdir*() function shall mark for update the *st_atime* field of the directory each time the directory is actually read.

If {_POSIX_THREAD_SAFE_FUNCTIONS} is defined:

> The *readdir_r*() function shall initialize the *dirent* structure referenced by *entry* to represent the directory entry at the current position in the directory stream referred to by *dirp*, store a pointer to this structure at the location referenced by *result*, and position the directory stream at the next entry.
> The storage pointed to by *entry* shall be large enough for a *dirent* with the *d_name* member an array of *char* containing at least {NAME_MAX} plus one elements.
> Upon successful return, the pointer returned at **result* shall have the same value as the argument *entry*. Upon reaching the end of the directory stream, this pointer shall have the value **NULL**.
> The *readdir_r*() function shall not return directory entries containing empty names. It is unspecified whether entries are returned for dot or dot-dot.
> 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.

Otherwise:

> Either the implementation shall support the *readir_r*() function as described above or the *readir_r*() function shall not be provided.

The *rewinddir*() function resets the position of the directory stream to which *dirp* refers to the beginning of the directory. It also causes the directory stream to refer to the current state of the corresponding directory, as a call to *opendir*() would have done. It does not return a value.

If a file is removed from or added to the directory after the most recent call to *opendir*() or *rewinddir*(), whether a subsequent call to *readdir*() or *readdir_r*() returns an entry for that file is unspecified.

The *closedir*() function closes the directory stream referred to by *dirp* and returns a value of zero if successful. Otherwise, it returns –1 indicating an error. Upon return, the value of *dirp* may no longer point to an accessible object of type *DIR*. If a file descriptor is used to implement type *DIR*, that file descriptor shall be closed.

If the *dirp* argument passed to any of these functions does not refer to a currently open directory stream, the effect is undefined.

The result of using a directory stream after one of the *exec* family of functions is undefined. After a call to the *fork*() function, either the parent or the child (but not both) may continue processing the directory stream using *readdir*() or *rewinddir*() or both. If both the parent and child processes use these functions, the result is undefined. Either or both processes may use *closedir*().

5.1.2.3 Returns

Upon successful completion, *opendir*() returns a pointer to an object of type *DIR*. Otherwise, a value of **NULL** is returned and *errno* is set to indicate the error.

Upon successful completion, *readdir*() returns a pointer to an object of type *struct dirent*. When an error is encountered, a value of **NULL** is returned and *errno* is set to indicate the error. When the end of the directory is encountered, a value of **NULL** is returned and *errno* is unchanged by this function call.

If successful, the *readdir_r*() function shall return zero. Otherwise, an error number shall be returned to indicate the error.

Upon successful completion, *closedir*() returns a value of zero. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

5.1.2.4 Errors

If any of the following conditions occur, the *opendir*() function shall return –1 and set *errno* to the corresponding value:

[EACCES]	Search permission is denied for a component of the path prefix of *dirname*, or read permission is denied for the directory itself.
[ENAMETOOLONG]	The length of the *dirname* argument exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.
[ENOENT]	The named directory does not exist, or *dirname* points to an empty string.
[ENOTDIR]	A component of *dirname* is not a directory.

For each of the following conditions, if the condition is detected, the *opendir*() function shall return –1 and set *errno* to the corresponding value:

[EMFILE]	Too many file descriptors are currently open for the process.

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

For each of the following conditions, if the condition is detected, the *readdir*() function shall return –1 and set *errno* to the corresponding value:

[EBADF]	The *dirp* argument does not refer to an open directory stream.

For each of the following conditions, if the condition is detected, the *readdir_r*() function shall return the corresponding error number:

[EBADF]	The *dirp* argument does not refer to an open directory stream.

For each of the following conditions, if the condition is detected, the *closedir*() function shall return –1 and set *errno* to the corresponding value:

[EBADF]	The *dirp* argument does not refer to an open directory stream.

5.1.2.5 Cross-References

`<dirent.h>`, 5.1.1; *exec*, 3.1.2.

5.2 Working Directory

5.2.1 Change Current Working Directory

Function: *chdir*()

5.2.1.1 Synopsis

```
int chdir(const char *path);
```

5.2.1.2 Description

The *path* argument points to the pathname of a directory. The *chdir*() function causes the named directory to become the current working directory, that is, the starting point for path searches of pathnames not beginning with slash.

If the *chdir*() function fails, the current working directory shall remain unchanged by this function call.

5.2.1.3 Returns

Upon successful completion, a value of zero is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

5.2.1.4 Errors

If any of the following conditions occur, the *chdir*() function shall return –1 and set *errno* to the corresponding value:

[EACCES]	Search permission is denied for any component of the path-name.
[ENAMETOOLONG]	The *path* argument exceeds {PATH_MAX} in length, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.
[ENOTDIR]	A component of the pathname is not a directory.

[ENOENT]	The named directory does not exist or *path* is an empty string.

5.2.1.5 Cross-References

getcwd(), 5.2.2.

5.2.2 Get Working Directory Pathname

Function: *getcwd*()

5.2.2.1 Synopsis

```
char *getcwd(char *buf, size_t size);
```

5.2.2.2 Description

The *getcwd*() function copies an absolute pathname of the current working directory to the array of *char* pointed to by the argument *buf* and returns a pointer to the result. The *size* argument is the size in bytes of the array of *char* pointed to by the *buf* argument. If *buf* is a **NULL** pointer, the behavior of *getcwd*() is undefined.

5.2.2.3 Returns

If successful, the *buf* argument is returned. A **NULL** pointer is returned if an error occurs and the variable *errno* is set to indicate the error. The contents of *buf* after an error are undefined.

5.2.2.4 Errors

If any of the following conditions occur, the *getcwd*() function shall return a value of **NULL** and set *errno* to the corresponding value:

[EINVAL]	The *size* argument is zero.
[ERANGE]	The *size* argument is greater than zero but smaller than the length of the pathname plus 1.

For each of the following conditions, if the condition is detected, the *getcwd*() function shall return a value of **NULL** and set *errno* to the corresponding value:

[EACCES]	Read or search permission was denied for a component of the pathname.

5.2.2.5 Cross-References

chdir(), 5.2.1.

5.3 General File Creation

5.3.1 Open a File

Function: *open*()

5.3.1.1 Synopsis

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
int open(const char *path, int oflag, ...);
```

5.3.1.2 Description

The *open*() function establishes the connection between a file and a file descriptor. It creates 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 a 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 does not share it with any other process in the system. The file offset shall be set to the beginning of the file. The FD_CLOEXEC file descriptor flag associated with the new file descriptor shall be cleared. The file status flags and file access modes of the open file description shall be set according to the value of *oflag*. The value of *oflag* is the bitwise inclusive OR of values from the following list. See 6.5.1 for the definitions of the symbolic constants. Applications shall specify exactly one of the first three values (file access modes) below in the value of *oflag*:

O_RDONLY	Open for reading only.
O_WRONLY	Open for writing only.
O_RDWR	Open for reading and writing. The result is undefined if this flag is applied to a FIFO.

Any combination of the remaining flags may be specified in the value of *oflag*:

O_APPEND	If set, the file offset shall be set to the end of the file prior to each write.
O_CREAT	This option requires a third argument, *mode*, which is of type *mode_t*. If the file exists, this flag has no effect, except as noted under O_EXCL below. Otherwise, the file is created; the file's user ID shall be set to the effective user ID of the process; the file's group ID shall be set to the group ID of the directory in which the file is being created or to the effective group ID of the process. The file permission bits (see 5.6.1) shall be set to the value of *mode* except those set in the file mode creation mask of the process (see 5.3.3). When bits in *mode* other than the file permission bits are set, the effect is unspecified. The *mode* argument does not affect whether the file is opened for reading, for writing, or for both.
O_DSYNC	Write I/O operations on the file descriptor complete as defined by synchronized I/O data integrity completion.
O_EXCL	If O_EXCL and O_CREAT 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 processes executing *open*() naming the same filename in the same directory with O_EXCL and O_CREAT set. 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, the *open*() function shall not cause the terminal device to become the controlling terminal for the process (see 7.1.1.3).

O_NONBLOCK

1) When opening a FIFO with O_RDONLY or O_WRONLY set:
   - a) 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.
   - b) 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.
2) When opening a block special or character special file that supports nonblocking opens:
   - a) If O_NONBLOCK is set:

The *open*() shall return without waiting for the device to be ready or available. Subsequent behavior of the device is device-specific.

b) If O_NONBLOCK is clear:
The *open*() shall block the calling thread until the device is ready or available before returning.

3) Otherwise, the behavior of O_NONBLOCK is unspecified.

O_RSYNC	Read I/O operations on the file descriptor 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 complete as defined by synchronized I/O data integrity completion. If both O_SYNC and O_RSYNC are set in *oflag*, all I/O operations on the file descriptor complete as defined by synchronized I/O file integrity completion.
O_SYNC	Write I/O operations on the file descriptor complete as defined by synchronized I/O file integrity completion.
O_TRUNC	If the file exists and is a regular file, and the file is successfully opened O_RDWR or O_WRONLY, it shall be truncated to zero length and the mode and owner shall be unchanged by this function call. O_TRUNC shall have no effect on FIFO special files or terminal device files. Its effect on other file types is implementation defined. The result of using O_TRUNC with O_RDONLY is undefined.

If O_CREAT is set and the file did not previously exist, upon successful completion the *open*() function shall mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the file and the *st_ctime* and *st_mtime* fields of the parent directory.

If both the O_SYNC and O_DSYNC flags are set, the effect is as if only the O_SYNC flag was set.

If O_TRUNC is set and the file did previously exist, upon successful completion the *open*() function shall mark for update the *st_crime* and *st_mtime* fields of the file.

5.3.1.3 Returns

Upon successful completion, the function shall open the file and return a nonnegative integer representing the lowest numbered unused file descriptor. Otherwise, it shall return –1 and shall set *errno* to indicate the error. No files shall be created or modified if the function returns –1.

5.3.1.4 Errors

If any of the following conditions occur, the *open*() function shall return –1 and set *errno* to the corresponding value:

[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]	The *open*() operation was interrupted by a signal.
[EINVAL]	This implementation does not support synchronized I/O for this file.
[EISDIR]	The named file is a directory, and the *oflag* argument specifies write or read/write access.
[EMFILE]	Too many file descriptors are currently in use by this process.
[ENAMETOOLONG]	The length of the *path* string exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.

[ENFILE]	Too many files are 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.
[ENOSPC]	The directory or file system that would contain the new file cannot be extended.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENXIO]	O_NONBLOCK is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading.
[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.

5.3.1.5 Cross-References

close(), 6.3.1; *creat*(), 5.3.2; *dup*(), 6.2.1; *exec*, 3.1.2; *fcntl*(), 6.5.2; `<fcntl.h>`, 6.5.1; *lseek*(), 6.5.3; *read*(), 6.4.1; `<signal.h>`, 3.3.1.1; *star*(), 5.6.2; `<sys/stat.h>`, 5.6.1; *write*(), 6.4.2; *umask*(), 5.3.3; 3.3.1.4.

5.3.2 Create a New File or Rewrite an Existing One

Function: *creat*()

5.3.2.1 Synopsis

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat(const char *path, mode_t mode);
```

5.3.2.2 Description

The function call:

```
      creat(path, mode);
```

is equivalent to:

```
      open(path, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

5.3.2.3 Cross-References

open(), 5.3.1; `<sys/lstat.h>`, 5.6.1.

5.3.3 Set File Creation Mask

Function: *umask*()

5.3.3.1 Synopsis

```
#include <sys/types.h>
#include <sys/stat.h>
mode_t umask(mode_t cmask);
```

5.3.3.2 Description

The *umask*() routine sets the file mode creation mask of the process to *cmask* and returns the previous value of the mask. Only the file permission bits (see 5.6.1) of *cmask* are used; the meaning of the other bits is implementation defined.

The file mode creation mask of the process is used during *open*(), *creat*(), *mkdir*(), and *mkfifo*() calls to turn off permission bits in the *mode* argument supplied. Bit positions that are set in *cmask* are cleared in the mode of the created file.

5.3.3.3 Returns

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 that returned value as *cmask* shall leave the state of the mask the same as its state before the first call, including any unspecified (by this part of ISO/IEC 9945) use of those bits.

5.3.3.4 Errors

The *umask*() function is always successful, and no return value is reserved to indicate an error.

5.3.3.5 Cross-References

chmod(), 5.6.4; *creat*(), 5.3.2; *mkdir*(), 5.4.1; *mkfifo*(), 5.4.2; *open*(), 5.3.1; `<sys/stat.h>`, 5.6.1.

5.3.4 Link to a File

Function: *link*()

5.3.4.1 Synopsis

```
int link(const char *existing, const char *new);
```

5.3.4.2 Description

The argument *existing* points to a pathname naming an existing file. The argument *new* points to a pathname naming the new directory entry to be created. Implementations may support linking of files across file systems. The *link*() function shall atomically create a new link for the existing file and increment the link count of the file by one.

If the *link*() function fails, no link shall be created, and the link count of the file shall remain unchanged by this function call.

The *existing* argument shall not name a directory unless the user has appropriate privileges and the implementation supports using *link*() on directories.

The implementation may require that the calling process has permission to access the existing file.

Upon successful completion, the *link*() function 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 are marked for update.

5.3.4.3 Returns

Upon successful completion, *link*() shall return a value of zero. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

5.3.4.4 Errors

If any of the following conditions occur, the *link*() function shall return −1 and set *errno* to the corresponding value:

[EACCES]	A component of either path prefix denies search permission; or the requested link requires writing in a directory with a mode that denies write permission; or the calling process does not have permission to access the existing file, and this is required by the implementation.
[EEXIST]	The link named by *new* exists.
[EMLINK]	The number of links to the file named by *existing* would exceed {LINK_MAX}.
[ENAMETOOLONG]	The length of the *existing* or *new* string exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.
[ENOENT]	A component of either path prefix does not exist, the file named by *existing* does not exist, or either *existing* or *new* points to an empty string.
[ENOSPC]	The directory that would contain the link cannot be extended.
[ENOTDIR]	A component of either path prefix is not a directory.
[EPERM]	The file named by *existing* is a directory, and either the calling process does not have appropriate privileges or the implementation prohibits using *link*() on directories.
[EROFS]	The requested link requires writing in a directory on a read-only file system.
[EXDEV]	The link named by *new* and the file named by *existing* are on different file systems, and the implementation does not support links between file systems.

5.3.4.5 Cross-References

rename(), 5.5.3; *unlink*(), 5.5.1.

5.4 Special File Creation

5.4.1 Make a Directory

Function: *mkdir*()

5.4.1.1 Synopsis

```
#include <sys/types.h>
#include <sys/stat.h>
int mkdir(const char *path, mode_t mode);
```

5.4.1.2 Description

The *mkdir*() routine creates a new directory with name *path*. The file permission bits of the new directory are initialized from *mode*. The file permission bits of the *mode* argument are modified by the file creation mask of the process (see 5.3.3). When bits in *mode* other than the file permission bits are set, the meaning of these additional bits is implementation defined.

The owner ID of the directory is set to the effective user ID of the process. The directory's group ID shall be set to the group ID of the directory in which the directory is being created or to the effective group ID of the process.

The newly created directory shall be an empty directory.

Upon successful completion, the *mkdir*() function shall mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the directory. Also, the *st_ctime* and *st_mtime* fields of the directory that contains the new entry are marked for update.

5.4.1.3 Returns

A return value of zero indicates success. A return value of –1 indicates that an error has occurred, and an error code is stored in *errno*. No directory shall be created if the return value is –1.

5.4.1.4 Errors

If any of the following conditions occur, the *mkdir*() function shall return –1 and set *errno* to the corresponding value:

[EACCES]	Search permission is denied on a component of the path prefix, or write permission is denied on the parent directory of the directory to be created.
[EEXIST]	The named file exists.
[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} while {_POSIX_NO_TRUNC} is in effect.
[ENOENT]	A component of the path prefix does not exist, or the *path* argument points to 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 of the directory being created resides on a read-only file system.

5.4.1.5 Cross-References

chmod(), 5.6.4; *stat*(), 5.6.2; `<sys/stat.h>`, 5.6.1; *umask*(), 5.3.3.

5.4.2 Make a FIFO Special File

Function: *mkfifo*()

5.4.2.1 Synopsis

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *path, mode_t mode);
```

5.4.2.2 Description

The *mkfifo*() routine creates a new FIFO special file named by the pathname pointed to by *path*. The file permission bits of the new FIFO are initialized from *mode*. The file permission bits of the *mode* argument are modified by the file creation mask of the process (see 5.3.3). When bits in *mode* other than the file permission bits are set, the effect is implementation defined.

The owner ID of the FIFO shall be set to the effective user ID of the process. The group ID of the FIFO shall be set to the group ID of the directory in which the FIFO is being created or to the effective group ID of the process.

Upon successful completion, the *mkfifo*() function 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 are marked for update.

5.4.2.3 Returns

Upon successful completion, a value of zero is returned. Otherwise, a value of –1 is returned, no FIFO is created, and *errno* is set to indicate the error.

5.4.2.4 Errors

If any of the following conditions occur, the *mkfifo*() function shall return –1 and set *errno* to the corresponding value:

[EACCES]	Search permission is denied on a component of the path prefix, or write permission is denied on the parent directory of the file to be created.
[EEXIST]	The named file already exists.
[ENAMETOOLONG]	The length of the *path* string exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.
[ENOENT]	A component of the path prefix does not exist, or the *path* argument points to 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.

5.4.2.5 Cross-References

chmod(), 5.6.4; *exec*, 3.1.2; *pipe*(), 6.1.1; *stat*(), 5.6.2; `<sys/stat.h>`, 5.6.1; *umask*(), 5.3.3.

5.5 File Removal

5.5.1 Remove Directory Entries

Function: *unlink*()

5.5.1.1 Synopsis

```
int unlink(const char *path);
```

5.5.1.2 Description

The *unlink*() function shall remove the link named by the pathname pointed to by *path* and decrement the link count of the file referenced by the link.

When the link count of the file becomes zero 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 have been closed.

The *path* argument shall not name a directory unless the process has appropriate privileges and the implementation supports using *unlink*() on directories. Applications should use *rmdir*() to remove a directory.

Upon successful completion, the *unlink*() function shall mark for update the *st_ctime* and *st_mtime* fields of the parent directory. Also, if the link count of the file is not zero, the *st_ctime* field of the file shall be marked for update.

5.5.1.3 Returns

Upon successful completion, a value of zero shall be returned. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error. If −1 is returned, the named file shall not be changed by this function call.

5.5.1.4 Errors

If any of the following conditions occur, the *unlink*() function shall return −1 and set *errno* to the corresponding value:

[EACCES]	Search permission is denied for a component of the path prefix, or write permission is denied on the directory containing the link to be removed.
[EBUSY]	The directory named by the *path* argument cannot be unlinked because it is being used by the system or another process and the implementation considers this to be an error.
[ENAMETOOLONG]	The length of the *path* argument exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.
[ENOENT]	The named file does not exist, or the *path* argument points to 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.
[EROFS]	The directory entry to be unlinked resides on a read-only file system.

5.5.1.5 Cross-References

close(), 6.3.1; *link*(), 5.3.4; *open*(), 5.3.1; *rename*(), 5.5.3; *rmdir*(), 5.5.2.

5.5.2 Remove a Directory

Function: *rmdir*()

5.5.2.1 Synopsis

```
int rmdir(const char *path);
```

5.5.2.2 Description

The *rmdir*() function removes a directory whose name is given by *path*. The directory shall be removed only if it is an empty directory.

If the named directory is the root directory or the current working directory of any process, it is unspecified whether the function succeeds or whether it fails and sets *errno* to [EBUSY].

If the link count of the directory becomes zero and no process has the directory open, the space occupied by the directory shall be freed and the directory shall no longer be accessible. If one or more processes have the directory open when the last link is removed, the dot and dot-dot entries, if present, are removed before *rmdir*() returns and no new entries may be created in the directory, but the directory is not removed until all references to the directory have been closed.

Upon successful completion, the *rmdir*() function shall mark for update the *st_ctime* and *st_mtime* fields of the parent directory.

5.5.2.3 Returns

Upon successful completion, a value of zero shall be returned. Otherwise, a value of –1 shall be returned and *errno* shall be set to indicate the error. If –1 is returned, the named directory shall not be changed by this function call.

5.5.2.4 Errors

If any of the following conditions occur, the *rmdir*() function shall return –1 and set *errno* to the corresponding value:

[EACCES]	Search permission is denied on a component of the path prefix, or write permission is denied on the parent directory of the directory to be removed.
[EBUSY]	The directory named by the *path* argument cannot be removed because it is being used by another process and the implementation considers this to be an error.
[EEXIST] or [ENOTEMPTY]	The *path* argument names a directory that is not an empty directory.
[ENAMETOOLONG]	The length of the *path* argument exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.
[ENOENT]	The *path* argument names a nonexistent directory or points to an empty string.
[ENOTDIR]	A component of the path is not a directory.
[EROFS]	The directory entry to be removed resides on a read-only file system.

5.5.2.5 Cross-References

mkdir(), 5.4.1; *unlink*(), 5.5.1.

5.5.3 Rename a File

Function: *rename*()

5.5.3.1 Synopsis

```
int rename(const char *old, const char *new);
```

5.5.3.2 Description

The *rename*() function changes the name of a file. The *old* argument points to the pathname of the file to be renamed. The *new* argument points to the new pathname of the file.

If the *old* argument and the *new* argument both refer to links to the same existing file, the *rename*() function shall return successfully and perform no other action.

If the *old* argument points to the pathname of a file that is not a directory, the *new* argument shall not point to the pathname of a directory. If the link named by the *new* argument exists, it shall be removed and *old* renamed to *new*. In this case, a link named *new* shall exist throughout the renaming operation and shall refer either to the file referred to by *new* or *old* before the operation began. Write access permission is required for both the directory containing *old* and the directory containing *new*.

If the *old* argument points to the pathname of a directory, the *new* argument shall not point to the pathname of a file that is not a directory. If the directory named by the *new* argument exists, it shall be removed and *old* renamed to *new*. In this case, a link named *new* shall exist throughout the renaming operation and shall refer either to the file referred to by *new* or *old* before the operation began. Thus, if *new* names an existing directory, it shall be required to be an empty directory.

The *new* pathname shall not contain a path prefix that names *old*. Write access permission is required for the directory containing *old* and the directory containing *new*. If the *old* argument points to the pathname of a directory, write access permission may be required for the directory named by *old*, and, if it exists, the directory named by *new*.

If the link named by the *new* argument exists and the link count of the file becomes zero when it is removed and no process has the file open, the space occupied by the file shall be freed and the file shall no longer be accessible. If one or more processes have the file open when the last link is removed, the link shall be removed before *rename*() returns, but the removal of the file contents shall be postponed until all references to the file have been closed.

Upon successful completion, the *rename*() function shall mark for update the *st_ctime* and *st_mtime* fields of the parent directory of each file.

5.5.3.3 Returns

Upon successful completion, a value of zero shall be returned. Otherwise, a value of –1 shall be returned and *errno* shall be set to indicate the error. If –1 is returned, neither the file named by *old* nor the file named by *new*, if either exists, shall be changed by this function call.

5.5.3.4 Errors

If any of the following conditions occur, the *rename*() function shall return –1 and set *errno* to the corresponding value:

[EACCES]	A component of either path prefix denies search permission, or one of the directories containing *old* or *new* denies write permissions, or write permission is required and is denied for a directory pointed to by the *old* or *new* arguments.
[EBUSY]	The directory named by *old* or *new* cannot be renamed because it is being used by the system or another process and the implementation considers this to be an error.
[EEXIST] or [ENOTEMPTY]	
	The link named by *new* is a directory containing entries other than dot and dot-dot.
[EINVAL]	The *new* directory pathname contains a path prefix that names the *old* directory.
[EISDIR]	The *new* argument points to a directory, and the *old* argument points to a file that is not a directory.
[ENAMETOOLONG]	
	The length of the *old* or *new* argument exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.
[EMLINK]	The file named by *old* is a directory, and the link count of the parent directory of *new* would exceed {LINK_MAX}.
[ENOENT]	The link named by the *old* argument does not exist, or either *old* or *new* points to an empty string.
[ENOSPC]	The directory that would contain *new* cannot be extended.
[ENOTDIR]	A component of either path prefix is not a directory, or the *old* argument names a directory and the *new* argument names a nondirectory file.
[EROFS]	The requested operation requires writing in a directory on a read-only file system.

[EXDEV] The links named by *new* and *old* are on different file systems, and the implementation does not support links between file systems.

5.5.3.5 Cross-References

link(), 5.3.4; *rmdir*(), 5.5.2; *unlink*(), 5.5.1.

5.6 File Characteristics

5.6.1 File Characteristics: Header and Data Structure

The header `<sys/stat.h>` defines the structure *stat*, which includes the members shown in Table 5.1, returned by the functions *stat*() and *fstat*().

Table 5.1—*stat* Structure

Member Type	Member Type	Description
mode_t	*st_mode*	File mode (see 5.6.1.2).
ino_t	*st_ino*	File serial number.
dev_t	*st_dev*	ID of device containing this file.
nlink_t	*st_nlink*	Number of links.
uid_t	*st_uid*	User ID of the owner of the file.
gid_t	*st_gid*	Group ID of the group of the file.
off_t	*st_size*	For regular files, the file size in bytes. For other file types, the use of this field is unspecified.
time_t	*st_atime*	Time of last access.
time_t	*st_mtime*	Time of last data modification.
time_t	*st_ctime*	Time of last file status change.

NOTE — File serial number and device ID taken together uniquely identify the file within the system.

All of the described members shall appear in the *stat* structure. The structure members *st_mode*, *st_ino*, *st_dev*, *st_uid*, *st_gid*, *st_atime*, *st_ctime*, and *st_mtime* shall have meaningful values for all file types defined in this part of ISO/IEC 9945. The value of the member *st_nlink* shall be set to the number of links to the file.

5.6.1.1 `<sys/stat.h>` File Types

The following macros shall test whether a file is of the specified type. The value *m* supplied to the macros is the value of *st_mode* from a *stat* structure. The macro evaluates to a nonzero value if the test is true, zero if the test is false.

S_ISDIR(*m*)	Test macro for a directory file.
S_ISCHR(*m*)	Test macro for a character special file.
S_ISBLK(*m*)	Test macro for a block special file.
S_ISREG(*m*)	Test macro for a regular file.
S_ISFIFO*(m)*	Test macro for a pipe or a FIFO special file.

The implementation may implement message queues, semaphores, or shared memory objects as distinct file types. The following macros shall test whether a file is of the specified type. The value of the *buf* argument supplied to the macros is a pointer to a *stat* structure. The macro shall evaluate to a nonzero value if the specified object is implemented as a distinct file type and the specified file type is contained in the *stat* structure referenced by *buf*. Otherwise, the macro shall evaluate to zero.

S_TYPEISMQ(*buf*)Test macro for a message queue

S_TYPEISSEM(*buf*)Test macro for a semaphore

S_TYPEISSHM(*buf*)Test macro for a shared memory object

5.6.1.2 `<sys/stat.h>` File Modes

The file modes portion of values of type *mode_t*, such as the *st_mode* value, are bit-encoded with the following masks and bits:

S_IRWXU	Read, write, search (if a directory), or execute (otherwise) permissions mask for the file owner class.	
	S_IRUSR	Read permission bit for the file owner class.
	S_IWUSR	Write permission bit for the file owner class.
	S_IXUSR	Search (if a directory) or execute (otherwise)permissions bit for the file owner class.
S_IRWXG	Read, write, search (if a directory), or execute (otherwise) permissions mask for the file group class.	
	S_IRGRP	Read permission bit for the file group class.
	S_IWGRP	Write permission bit for the file group class.
	S_IXGRP	Search (if a directory) or execute (otherwise) permissions bit for the file group class.
S_IRWXO	Read, write, search (if a directory), or execute (otherwise) permissions mask for the file other class.	
	S_IROTH	Read permission bit for the file other class.
	S_IWOTH	Write permission bit for the file other class.
	S_IXOTH	Search (if a directory) or execute (otherwise) permissions bit for the file other class.
S_ISUID	Set user ID on execution. The effective user ID of the process shall be set to that of the owner of the file when the file is run as a program (see *exec*). On a regular file, this bit should be cleared on any write.	
S_ISGID	Set group ID on execution. Set effective group ID on the process to the group of the file when the file is run as a program (see *exec*). On a regular file, this bit should be cleared on any write.	

The bits defined by S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH, S_IWOTH, S_IXOTH, S_ISUID, and S_ISGID shall be unique. S_IRWXU shall be the bitwise inclusive OR of S_IRUSR, S_IWUSR, and S_IXUSR. S_IRWXG shall be the bitwise inclusive OR of S_IRGRP, S_IWGRP, and S_IXGRP. S_IRWXO shall be the bitwise inclusive OR of S_IROTH, S_IWOTH, and S_IXOTH. Implementations may OR other implementation-defined bits into S_IRWXU, S_IRWXG, and S_IRWXO, but they shall not overlap any of the other bits defined in this part of ISO/IEC 9945. The file permission bits are defined to be those corresponding to the bitwise inclusive OR of S_IRWXU, S_IRWXG, and S_IRWXO.

5.6.1.3 `<sys/stat.h>` Time Entries

The time-related fields of *struct stat* are as follows:

st_atime	Accessed file data, for example, *read*().
st_mtime	Modified file data, for example, *write*().
st_crime	Changed file status, for example, *chmod*().

These times are updated as described in B.2.3.5.

Times are given in seconds since the Epoch.

5.6.1.4 Cross-References

chmod(), 5.6.4; *chown*(), 5.6.5; *creat*(), 5.3.2; *exec*, 3.1.2; *link*(), 5.3.4; *mkdir*(), 5.4.1; *mkfifo*(), 5.4.2; *pipe*(), 6.1.1; *read*(), 6.4.1; *unlink*(), 5.5.1; *utime*(), 5.6.6; *write*(), 6.4.2; *remove*() [C Standard {2}].

5.6.2 Get File Status

Functions: *stat*(), *fstat*()

5.6.2.1 Synopsis

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *path, struct stat *buf);
int fstat(int fildes, struct stat *buf);
```

5.6.2.2 Description

The *path* argument points to a pathname naming a file. Read, write, or execute permission for the named file is not required, but all directories listed in the pathname leading to the file must be searchable. The *stat*() function obtains information about the named file and writes it to the area pointed to by the *buf* argument.

Similarly, the *fstat*() function obtains information about an open file known by the file descriptor *fildes*. If the Shared Memory Objects option is supported and *fildes* references a shared memory object, the implementation needs to update in the *stat* structure pointed to by the *buf* argument only the *st_uid*, *st_gid*, *st_size*, and *st_mode* fields, and only the S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits need be valid.

An implementation that provides additional or alternate file access control mechanisms may, under implementation-defined conditions, cause the *stat*() and *fstat*() functions to fail. In particular, the system may deny the existence of the file specified by *path*.

Both functions update any time-related fields, as described in B.2.3.5, before writing into the *stat* structure.

The *buf* is taken to be a pointer to a *stat* structure, as defined in the header `<sys/stat.h>`, into which information is placed concerning the file.

5.6.2.3 Returns

Upon successful completion, a value of zero shall be returned. Otherwise, a value of –1 shall be returned and *errno* shall be set to indicate the error.

5.6.2.4 Errors

If any of the following conditions occur, the *stat*() function shall return –1 and set *errno* to the corresponding value:

[EACCES]	Search permission is denied for a component of the path prefix.
[ENAMETOOLONG]	The length of the *path* argument exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.
[ENOENT]	The named file does not exist, or the *path argument* points to an empty string.
[ENOTDIR]	A component of the path prefix is not a directory.

If any of the following conditions occur, the *fstat*() function shall return –1 and set *errno* to the corresponding value:

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

5.6.2.5 Cross-References

creat(), 5.3.2; *dup*(), 6.2.1; *fcntl*(), 6.5.2; *open*(), 5.3.1; *pipe*(), 6.1.1; `<sys/stat.h>`, 5.6.1.

5.6.3 Check File Accessibility

Function: *access*()

5.6.3.1 Synopsis

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

5.6.3.2 Description

The *access*() function checks the accessibility of the file named by the pathname pointed to by the *path* argument for the file access permissions indicated by *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, and X_OK) or the existence test (F_OK). See 2.9.1 for the description of these symbolic constants.

If any access permission is to be checked, each shall be checked individually, as described in B.2.3.2. If the process has appropriate privileges, an implementation may indicate success for X_OK even if none of the execute file permission bits are set.

5.6.3.3 Returns

If the requested access is permitted, a value of zero shall be returned. Otherwise, a value of –1 shall be returned and *errno* shall be set to indicate the error.

5.6.3.4 Errors

If any of the following conditions occur, the *access*() function shall return –1 ανδ σετ *errno* to the corresponding value:

[EACCES] The permissions specified by *amode* are denied, or search permission is denied on a component of the path prefix.

[ENAMETOOLONG]

The length of the *path* argument exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.

[ENOENT] The *path* argument points to an empty string or to the name of a file that does not exist.

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

[EROFS] Write access was requested for a file residing on a read-only file system.

For each of the following conditions, if the condition is detected, the *access*() function shall return –1 and set *errno* to the corresponding value:

[EINVAL] An invalid value was specified for *amode*.

5.6.3.5 Cross-References

chmod(), 5.6.4; *stat*(), 5.6.2; `<unistd.h>`, 2.9.

5.6.4 Change File Modes

Function: *chmod*(), *fchmod*()

5.6.4.1 Synopsis

```
#include <sys/stat.h>
int chmod(const char *path, mode_t mode);
int fchmod(int fildes, mode_t mode);
```

5.6.4.2 Description

The *path* argument shall point to a pathname naming a file. If the effective user ID of the calling process matches the file owner or the calling process has appropriate privileges, the *chmod*() function shall set the S_ISUID, S_ISGID, and the file permission bits, as described in 5.6.1, of the named file from the corresponding bits in the *mode* argument. These bits define access permissions for the user associated with the file, the group associated with the file, and all others, as described in B.2.3.2. Additional implementation-defined restrictions may cause the S_ISUID and S_ISGID bits in *mode* to be ignored.

If at least one of {_POSIX_MAPPED_FILES} or {_POSIX_SHARED_MEMORY_OBJECTS} is defined:

> The *fchmod*() function similarly sets the S_ISUID, S_ISGID, and file permission bits of the open file known by the file descriptor *fildes*. If the Shared Memory Objects option is supported and *fildes* references a shared memory object, the *chmod*() function need affect only the S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits.
>
> NOTE — It is expected that a future revision will make this interface mandatory for conforming implementations and regular files. The text of the condition on {_POSIX_MAPPED_FILES} and {_POSIX_SHARED_MEMORY_OBJECTS} will be removed at the time the contemplated revision is approved. Since the existence of shared memory will remain optional, the semantics relating to it will remain conditional.

Otherwise:

> Either the implementation shall support the *fchmod*() function as described above or the *fchmod*() function shall fail.

If the calling process does not have appropriate privileges, if the group ID of the file does not match the effective group ID or one of the supplementary group IDs, if one or more of the S_IXUSR, S_IXGRP, or S_IXOTH bits of the file mode are set, and if the file is a regular file, bit S_ISGID (set group ID on execution) in the mode of the file shall be cleared upon successful return from *chmod*() or *fchmod*().

The effect on file descriptors for files open at the time of the *chmod*() or *fchmod*() function is implementation defined.

Upon successful completion, the *chmod*() or *fchmod*() function shall mark for update the *st_ctime* field of the file.

5.6.4.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of –1 shall be returned and *errno* shall be set to indicate the error. If –1 is returned, no change to the file mode shall have occurred.

5.6.4.4 Errors

If any of the following conditions occur, the *chmod*() function shall return –1 and set *errno* to the corresponding value:

[EACCES]	Search permission is denied on a component of the path prefix.
[ENAMETOOLONG]	The length of the *path* argument exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist or the *path* argument points to an empty string.
[EPERM]	The effective user ID does not match the owner of the file, and the calling process does not have the appropriate privileges.
[EROFS]	The named file resides on a read-only file system.

If any of the following conditions occur, the *fchmod*() function shall return –1 and set *errno* to the corresponding value:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[ENOSYS]	This implementation does not support the *fchmod*() function.
[EPERM]	The effective user ID does not match the owner of the file and the calling process does not have the appropriate privileges.
[EROFS]	The file resides on a read-only file system.

For each of the following conditions, if the condition is detected, the *fchmod*() function shall return –1 and set *errno* to the corresponding value:

[EINVAL]	The *fildes* argument refers to a pipe and the implementation disallows execution of *fchmod*() on a pipe.

5.6.4.5 Cross-References

chown(), 5.6.5; *mkdir*(), 5.4.1; *mkfifo*(), 5.4.2; *stat*(), 5.6.2; `<sys/stat.h>`, 5.6.1.

5.6.5 Change Owner and Group of a File

Function: *chown*()

5.6.5.1 Synopsis

```
#include <sys/types.h>
int chown(const char *path, uid_t owner, gid_t group);
```

5.6.5.2 Description

The *path* argument points to a pathname naming a file. The user ID and group ID of the named file are 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*:

1) Changing the owner is restricted to processes with appropriate privileges.
2) Changing the group is permitted to a process without appropriate privileges, but with an effective user ID equal to the user ID of the file, if and only if *owner* is equal to the user ID of the file and *group* is equal either to the effective group ID of the calling process or to one of its supplementary group IDs.

If the *path* argument refers to a regular file, the set-user-ID (.IOU Con S_ISUID) and set-group-ID (S_ISGID) bits of the file mode shall be cleared upon successful return from *chown*(), unless the call is made by a process with appropriate privileges, in which case it is implementation defined whether those bits are altered. If the *chown*() function is successfully invoked on a file that is not a regular file, these bits may be cleared. These bits are defined in 5.6.1.

Upon successful completion, the *chown*() function shall mark for update the *st_ctime* field of the file.

5.6.5.3 Returns

Upon successful completion, a value of zero shall be returned. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error. If −1 is returned, no change shall be made in the owner and group of the file.

5.6.5.4 Errors

If any of the following conditions occur, the *chown*() function shall return −1 and set *errno* to the corresponding value:

[EACCES]	Search permission is denied on a component of the path prefix.
[ENAMETOOLONG]	The length of the *path* argument exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist, or the *path* argument points to an empty string.

[EPERM]	effective user ID does not match the owner of the file, or the calling process does not have appropriate privileges and {_POSIX_CHOWN_RESTRICTED} indicates that such privilege is required.
[EROFS]	The named file resides on a read-only file system.

For each of the following conditions, if the condition is detected, the *chown*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL]	The owner or group ID supplied is invalid and not supported by the implementation.

5.6.5.5 Cross-References

chmod(), 5.6.4; `<sys/stat.h>`, 5.6.1.

5.6.6 Set File Access and Modification Times

Function: *utime*()

5.6.6.1 Synopsis

```
#include <sys/types.h>
#include <utime.h>
int utime(const char *path, const struct utimbuf *times);
```

5.6.6.2 Description

The argument *path* points to a pathname naming a file. The *utime*() function sets the access and modification times of the named file.

If the *times* argument is **NULL**, the access and modification times of the file are set to the current time. The effective user ID of the process must match the owner of the file, or the process must have write permission to the file or appropriate privileges, to use the *utime*() function in this manner.

If the *times* argument is not **NULL**, it is interpreted as a pointer to a *utimbuf* structure, and the access and modification times are set to the values contained in the designated structure. Only the owner of the file and processes with appropriate privileges shall be permitted to use the *utime*() function in this way.

The *utimbuf* structure is defined by the header `<utime.h>` and includes the following members:

Member Type	**Member Name**	**Description**
time_t	*actime*	Access time
time_t	*modtime*	Modification time

The times in the *utimbuf* structure are measured in seconds since the Epoch.

Implementations may add extensions as permitted in 1.3.1.1, item (2). Adding extensions to this structure, which might change the behavior of the application with respect to this standard when those fields in the structure are uninitialized, also requires that the extensions be enabled as required by 1.3.1.1.

Upon successful completion, the *utime*() function shall mark for update the *st_ctime* field of the file.

5.6.6.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of −1 shall be returned, *errno* is set to indicate the error, and the file times shall not be affected.

5.6.6.4 Errors

If any of the following conditions occur, the *utime*() function shall return −1 ανδ σετ *errno* to the corresponding value:

[EACCES] Search permission is denied by a component of the path prefix, or the *times* argument is **NULL** and the effective user ID of the process does not match the owner of the file and write access is denied.

[ENAMETOOLONG]

The length of the *path* argument exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.

[ENOENT] The named file does not exist or the *path* argument points to an empty string.

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

[EPERM] The *times* argument is not **NULL**, the effective user ID of the calling process 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 *named* file resides on a read-only file system.

5.6.6.5 Cross-References

`<sys/stat.h>`, 5.6.1.

5.6.7 Truncate a File to a Specified Length

Function: *ftruncate*()

5.6.7.1 Synopsis

```
#include <unistd.h>
int ftruncate(int fildes, off_t length);
```

5.6.7.2 Description

If at least one of {_POSIX_MAPPED_FILES} or {_POSIX_SHARED_MEMORY_OBJECTS} is defined:

The *ftruncate*() function shall cause the regular file known by the file descriptor *fildes*, which must be open for writing, to be truncated to *length*. If the size of the file previously exceeded *length*, the extra data shall be discarded. If the file previously was smaller than this size, it is unspecified whether the file is changed or its size increased. If the file is extended, the extended area shall appear as if it were zero-filled. If *fildes* references a shared memory object, *ftruncate*() sets the size of the shared memory object to *length*. If the file is not a regular file or a shared memory object, the result is unspecified.
If the effect of *ftruncate*() is to decrease the size of a file or shared memory object and whole pages beyond the new end were previously mapped, then the whole pages beyond the new end shall be discarded. If the Memory Protection option is supported, references to such pages shall result in the generation of a SIGBUS signal; otherwise, the result of such references is undefined.
The value of the seek pointer shall not be modified by a call to *ftruncate*().

Upon successful completion, the *ftruncate*() function shall mark for update the *st_ctime* and *st_mtime* fields of the file. If the *ftruncate*() function is unsuccessful, the file is unaffected.

Otherwise:

Either the implementation shall support the *ftruncate*() function as described above or the *ftruncate*() function shall fail.

NOTE — It is expected that a future revision will make this interface mandatory for conforming implementations and regular files. The text of the support condition will be removed at the time the contemplated revision is approved. Since the existence of file mappings and shared memory will remain optional, the semantics relating to them will remain conditional.

5.6.7.3 Returns

Upon successful completion, the *ftruncate*() function shall return zero. Otherwise, it shall return −1 and set *errno* to indicate the error.

5.6.7.4 Errors

If any of the following conditions occur, the *ftruncate*() function shall return −1 and set *errno* to the corresponding value:

[EBADF]	The *fildes* argument is not a valid file descriptor open for writing.
[EINVAL]	The *fildes* argument does not refer to a file on which this operation is possible.
[EROFS]	The file resides on a read-only file system.

5.6.7.5 Cross-References

mmap() 12.2.1; *open*(), 5.3.1; *shm_open*() 12.3.1.

5.7 Configurable Pathname Variables

5.7.1 Get Configurable Pathname Variables

Functions: *pathconf*(), *fpathconf*()

5.7.1.1 Synopsis

```
#include <unistd.h>
long pathconf(const char *path, int name);
long fpathconf(int fildes, int name);
```

5.7.1.2 Description

The *pathconf*() and *fpathconf*() functions provide a method for the application to 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. The implementation shall support all of the variables listed in Table 5.2 and may support others. The variables in Table 5.2 come from

`<limits.h>` or `<unistd.h>` and the symbolic constants, defined in `<unistd.h>`, that are the corresponding values used for *name*.

Table 5.2—Configurable Pathname Variables

Variable	*name* Value	Notes
{LINK_MAX}	{_PC_LINK_MAX}	(1)
{MAX_CANON}	{_PC_MAX_CANON}	(2)
{MAX_INPUT}	{_PC_MAX_INPUT}	(2)
{NAME_MAX}	{_PC_NAME_MAX}	(3), (4)
{PATH_MAX}	{_PC_PATH_MAX}	(4), (5)
{PIPE_BUF}	{_PC_PIPE_BUF}	(6)
{_POSIX_ASYNC_IO}	[_PC_ASYNC_IO}	(8)
{_POSIX_CHOWN_RESTRICTED}	{PC_CHOWN_RESTRICTED}	(7)
{_POSIX_NO_TRUNC}	{_PC_NO_TRUNC}	(3, 4)
{_POSIX_PRIO_IO}	{_PC_PRIO_IO}	(8)
{_POSIX_SYNC_IO}	{_PC_SYNC_IO}	(8)
{_POSIX_VDISABLE}	{_PC_VDISABLE}	(2)

NOTES:

1 — If *path* or *fildes* refers to a directory, the value returned applies 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 applies to the 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 is 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 a FIFO, the value returned applies to the referenced object itself. If *path* or *fildes* refers to a directory, the value returned applies to any FIFOs that exist 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 applies to any files defined in this part of ISO/IEC 9945 , 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.

5.7.1.3 Returns

If *name* is an invalid value, the *pathconf*() and *fpathconf*() functions shall return –1.

If the variable corresponding to *name* has no limit for the path or file descriptor, the *pathconf*() and *fpathconf*() functions 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 the appropriate privileges to query the file specified by *path*, or *path* does not exist, the *pathconf*() function shall return –1.

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, the *fpathconf*() function shall return –1.

Otherwise, the *pathconf*() and *fpathconf*() functions 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 described to the application when it was compiled with the implementation's `<limits.h>` or `<unistd.h>`.

5.7.1.4 Errors

If any of the following conditions occur, the *pathconf*() and *fpathconf*() functions shall return –1 and set *errno* to the corresponding value:

[EINVAL] The value of *name* is invalid.

For each of the following conditions, if the condition is detected, the *pathconf*() function shall return –1 and set *errno* to the corresponding value:

[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.

[ENAMETOOLONG]

The length of the *path* argument exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.

[ENOENT] The named file does not exist, or the *path* argument points to an empty string.

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

For each of the following conditions, if the condition is detected, the *fpathconf*() function shall return –1 and set *errno* to the corresponding value:

[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.

6. Input and Output Primitives

The functions in this section deal with input and output from files and pipes. Functions are also specified that deal with the coordination and management of file descriptors and I/O activity.

6.1 Pipes

6.1.1 Create an Inter-Process Channel

Function: *pipe*()

6.1.1.1 Synopsis

```
int pipe(int fildes [2]);
```

6.1.1.2 Description

The *pipe*() function shall create a pipe and place two file descriptors, one each into the arguments *fildes*[0] and *fildes*[1], that refer to the open file descriptions for the read and write ends of the pipe. Their integer values shall be the two lowest available at the time of the *pipe*() function call. The O_NONBLOCK and FD_CLOEXEC flags shall be clear on both file descriptors. [The *fcntl*() function can be used to set these flags.]

Data can be written to file descriptor *fildes*[1] and read from file descriptor *fildes*[0]. A read on file descriptor *fildes*[0] shall access the data written to file descriptor *fildes*[1] on a first-in-first-out basis.

A process has the pipe open for reading if it has a file descriptor open that refers to the read end, *fildes*[0]. A process has the pipe open for writing if it has a file descriptor open that refers to the write end, *fildes*[1].

Upon successful completion, the *pipe*() function shall mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the pipe.

6.1.1.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error.

6.1.1.4 Errors

If any of the following conditions occur, the *pipe*() function shall return −1 and set *errno* to the corresponding value:

[EMFILE]	More than {OPEN_MAX}-2 file descriptors are already in use by this process.
[ENFILE]	The number of simultaneously open files in the system would exceed a system-imposed limit.

6.1.1.5 Cross-References

fcntl(), 6.5.2; *open*(), 5.3.1; *read*(), 6.4.1; *write*(), 6.4.2.

6.2 File Descriptor Manipulation

6.2.1 Duplicate an Open File Descriptor

Functions: *dup*(), *dup2*()

6.2.1.1 Synopsis

```
int dup(int fildes);
int dup2(int fildes, int fildes2);
```

6.2.1.2 Description

The *dup*() and *dup2*() functions provide an alternate interface to the service provided by the *fcntl*() function 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:

1) If *fildes2* is negative or greater than or equal to {OPEN_MAX}, the *dup2*() function shall return –1 and *errno* shall be set to [EBADF].
2) If *fildes* is a valid file descriptor and is equal to *fildes2*, the *dup2*() function shall return *fildes2* without closing it.
3) If *fildes* is not a valid file descriptor, *dup2*() shall fail and not close *fildes2*.
4) The value returned shall be equal to the value of *fildes2* upon successful completion or shall be –1 upon failure.

6.2.1.3 Returns

Upon successful completion, the function shall return a file descriptor. Otherwise, a value of –1 shall be returned and *errno* shall be set to indicate the error.

6.2.1.4 Errors

If any of the following conditions occur, the *dup*() function shall return –1 and set *errno* to the corresponding value:

[EBADF]	The argument *fildes* is not a valid open file descriptor.
[EMFILE]	The number of file descriptors would exceed {OPEN_MAX}.

If any of the following conditions occur, the *dup2*() function shall return –1 and set *errno* to the corresponding value:

[EBADF]	The argument *fildes* 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.

6.2.1.5 Cross-References

close(), 6.3.1; *creat*(), 5.3.2; *exec*, 3.1.2; *fcntl*(), 6.5.2; *open*(), 5.3.1; *pipe*(), 6.1.1.

6.3 File Descriptor Deassignment

6.3.1 Close a File

Function: *close*()

6.3.1.1 Synopsis

```
int close(int fildes);
```

6.3.1.2 Description

The *close*() function shall deallocate (i.e., make available for return by subsequent *open*()s, etc., executed by the process) the file descriptor indicated by *fildes*. All outstanding record locks owned by the process on the file associated with the file descriptor shall be removed (that is, unlocked).

If the *close*() function 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.

When all file descriptors associated with a pipe or FIFO special file have been closed, any data remaining in the pipe or FIFO shall be discarded.

If the Asynchronous Input and Output option is supported:

> 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 cancelled, and which I/O operation may be cancelled upon *close*(), is implementation defined.

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 zero, when all file descriptors associated with the file have been closed, the space occupied by the file shall be freed and the file shall no longer be accessible.

If the Mapped Files or Shared Memory Objects option is supported:

> If a memory object remains referenced at the last close (i.e., a process has it mapped), then the entire contents of the memory object shall persist until the memory object becomes unreferenced. If this is the last close of a 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.

6.3.1.3 Returns

Upon successful completion, a value of zero shall be returned. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error.

6.3.1.4 Errors

If any of the following conditions occur, the *close*() function shall return −1 and set *errno* to the corresponding value:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINTR]	The *close*() function was interrupted by a signal.

6.3.1.5 Cross-References

creat(), 5.3.2; *dup*(), 6.2.1; *exec*, 3.1.2; *fcntl*(), 6.5.2; *fork*(), 3.1.1; *open*(), 5.3.1; *pipe*(), 6.1.1; *unlink*(), 5.5.1; 3.3.1.4.

6.4 Input and Output

6.4.1 Read from a File

Function: *read*()

6.4.1.1 Synopsis

`ssize_t read(int` *fildes*`, void` **buf*`, size_t` *nbyte*`);`

6.4.1.2 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*.

If *nbyte* is zero, the *read*() function shall return zero and have no other results.

On a regular file or other file capable of seeking, *read*() shall start at a position in the file given by the file offset associated with *fildes*. Before successful return from *read*(), the file offset shall be incremented by the number of bytes actually read.

On a file not capable of seeking, the *read*() shall start from the current position. The value of a file offset associated with such a file is undefined.

Upon successful completion, the *read*() function shall return the number of bytes actually read and placed in the buffer. This number shall never be greater than *nbyte*. The value returned may be less than *nbyte* if the number of bytes left in the 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, either it shall return −1 with *errno* set to [EINTR], or it shall return the number of bytes read. A *read*() from a pipe or FIFO shall never return with *errno* set to [EINTR] if it has transferred any data.

No data transfer shall occur past the current end-of-file. If the starting position is at or after the end-of-file, zero 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):

1) If no process has the pipe open for writing, *read*() shall return zero to indicate end-of-file.
2) If some process has the pipe open for writing and O_NONBLOCK is set, *read*() shall return −1 and set *errno* to [EAGAIN].
3) 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 nonblocking reads and has no data currently available:

1) If O_NONBLOCK is set, *read*() shall return −1 and set *errno* to [EAGAIN].
2) 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.

For any portion of a regular file, prior to the end-of-file, that has not been written, *read*() shall return bytes with value zero.

Upon successful completion where *nbyte* is greater than zero, the *read*() function shall mark for update the *st_atime field* of the file.

If the Synchronized Input and Output option is supported:

> If the O_DSYNC and O_RSYNC bits have been set, read I/O operations on the file descriptor complete as defined by synchronized I/O data integrity completion. If the O_SYNC and 0_RSYNC bits have been set, read I/O operations on the file descriptor complete as defined by synchronized I/O file integrity completion.

If the Shared Memory Objects option is supported:

> If *fildes* refers to a shared memory object, the result of the *read*() function is unspecified.

6.4.1.3 Returns

Upon successful completion, *read*() shall return an integer indicating the number of bytes actually read. Otherwise, *read*() shall return a value of –1 and set *errno* to indicate the error, and the content of the buffer pointed to by *buf* is indeterminate.

6.4.1.4 Errors

If any of the following conditions occur, the *read*() function shall return –1 ανδ σετ *errno* to the corresponding value:

[EAGAIN]	The O_NONBLOCK flag is set for the file descriptor and the process would be delayed in the read operation.
[EBADF]	The *fildes* argument is not a valid file descriptor open for reading.
[EINTR]	The read operation was interrupted by a signal, and either no data was transferred or the implementation does not report partial transfer for this file.
[EIO]	The implementation supports Job Control, the process is in a background process group and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the SIGTTIN signal or the process group of the process is orphaned. This error may also be generated when conditions unspecified by this part of ISO/IEC 9945 occur.

6.4.1.5 Cross-References

creat(), 5.3.2; *dup*(), 6.2.1; *fcntl*(), 6.5.2; *lseek*(), 6.5.3; *open*(), 5.3.1; *pipe*(), 6.1.1; 3.3.1.4; 7.1.1.

6.4.2 Write to a File

Function: *write*()

6.4.2.1 Synopsis

```
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

6.4.2.2 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*.

If *nbyte* is zero and the file is a regular file, the *write*() function shall return zero and have no other results. If *nbyte* is zero and the file is not a regular file, the results are unspecified.

On a regular file or other file capable of seeking, the actual writing of data shall proceed from the position in the file indicated by the file offset associated with *fildes*. Before successful return from *write*(), the file offset shall be incremented by the number of bytes actually written. On a regular file, if this incremented file offset is greater than the length of the file, the length of the file shall be set to this file offset.

On a file not capable of seeking, the *write*() shall start from the current position. The value of a file offset associated with such a file 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 be allowed between changing the file offset and the write operation.

If a *write*() requests that more bytes be written than there is room for (for example, 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 would return 20. The next write of a nonzero number of bytes would give a failure return (except as noted below).

Upon successful completion, the *write*() function shall return the number of bytes actually written to the file associated with *fildes*. This number shall never be greater than *nbyte*.

If a *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, either it shall return –1 with *errno* set to [EINTR], or it shall return the number of bytes written. A *write*() to a pipe or FIFO shall never return with *errno* set to [EINTR] if it has transferred any data and *nbyte* is less than or equal to {PIPE_BUF}.

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:

1) 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.
2) Any subsequent successful *write*() to the same byte position in the file shall overwrite that file data. The phrase "subsequent successful *write*()" in the previous sentence is intended to be viewed from a system perspective [i.e., *read*() followed by a systemwide subsequent *write*()].

Write requests to a pipe (or FIFO) handled in the same manner as write requests to a regular file, with the following exceptions:

1) There is no file offset associated with a pipe, hence each write request shall append to the end of the pipe.
2) 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.
3) If the O_NONBLOCK flag is clear, a write request may cause the thread to block, but on normal completion it shall return *nbyte*.

4) If the O_NONBLOCK flag is set, *write*() requests shall be handled differently, in the following ways:
   a) The *write*() function shall not block the process.
   b) A write request for {PIPE_BUF} or fewer bytes shall either:
      1) If there is sufficient space available in the pipe, transfer all the data and return the number of bytes requested.
      2) If there is not sufficient space available in the pipe, transfer no data and return –1 ωιτη *errno* set to [EAGAIN].
   c) A write request for more than {PIPE BUF} bytes shall either:
      1) 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 has been read, it shall transfer at least {PIPE_BUF} bytes.
      2) 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 nonblocking writes and cannot accept the data immediately:

1) If the O_NONBLOCK flag is clear, *write*() shall block the calling thread until the data can be accepted.
2) If the O_NONBLOCK flag is set, *write*() shall not block the process. If some data can be written without blocking the process, *write*() shall write what it can and return the number of bytes written. Otherwise, it shall return –1 and *errno* shall be set to [EAGAIN].

Upon successful completion where *nbyte* is greater than zero, the *write*() function shall mark for update the *st_ctime* and *st_mtime* fields of the file.

If the Synchronized Input and Output option is supported:

> If the O_DSYNC bit has been set, write I/O operations on the file descriptor 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 complete as defined by synchronized I/O file integrity completion.

If the Shared Memory Objects option is supported:

> If *fildes* refers to a shared memory object, the result of the *write*() function is unspecified.

6.4.2.3 Returns

Upon successful completion, *write*() shall return an integer indicating the number of bytes actually written. Otherwise, it shall return a value of –1 and set *errno* to indicate the error.

6.4.2.4 Errors

If any of the following conditions occur, the *write*() function shall return –1 and set *errno* to the corresponding value:

[EAGAIN]	The O_NONBLOCK flag is set for the file descriptor and the process would be delayed in the write operation.
[EBADF]	The *fildes* argument is not a valid file descriptor open for writing.
[EFBIG]	An attempt was made to write a file that exceeds an implementation-defined maximum file size.
[EINTR]	The write operation was interrupted by a signal, and either no data was transferred or the implementation does not report partial transfers for this file.
[EIO]	The implementation supports Job Control, the process is in a background process group and is attempting to write to its controlling terminal, TOSTOP is set, the process is neither ignoring nor

blocking SIGTTOU signals, and the process group of the process is orphaned. This error may also be generated when conditions unspecified by this part of ISO/IEC 9945 occur.

[ENOSPC]	There is no free space remaining on the device containing the file.
[EPIPE]	An attempt is made to write to a pipe (or FIFO) that is not open for reading by any process. A SIGPIPE signal shall also be sent to the process.

6.4.2.5 Cross-References

creat(), 5.3.2; *dup*(), 6.2.1; *fcntl*(), 6.5.2; *lseek*(), 6.5.3; *open*(), 5.3.1; *pipe*(), 6.1.1; 3.3.1.4.

6.5 Control Operations on Files

6.5.1 Data Definitions for File Control Operations

The header `<fcntl.h>` defines the following *requests* and *arguments* for the *fcntl*() and *open*() functions. The values within each of the tables within this clause (Table 6.1 through Table 6.7) shall be unique numbers. In addition, the values of the entries for *oflag* values, file status flags, and file access modes shall be unique.

6.5.2 File Control

Function: *fcntl*()

6.5.2.1 Synopsis

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fildes, int cmd, ...);
```

6.5.2.2 Description

The function *fcntl*() provides for control over open files. The argument *fildes* is a file descriptor.

The available values for *cmd* are defined in the header `<fcntl.h>` (see 6.5.1), which shall include:

F_DUPFD Return a new file descriptor that is the lowest numbered available (i.e., not already open) file descriptor greater than or equal to the third argument, *arg*, taken as an integer of type *int*. The new file descriptor refers to the same open file description as he original file descriptor and shares any locks.

The FD_CLOEXEC flag associated with the new file descriptor is cleared to keep the file open across calls to the *exec* family of functions.

Table 6.1—*cmd* Values for *fcntl*()

Constant	Description
F_DUPFD	Duplicate file descriptor.
F_GETFD	Get file descriptor flags.
F_GETLK	Get record locking information.
F_SETFD	Set file descriptor flags.
F_GETFL	Get file status flags.
F_SETFL	Set file status flags.
F_SETLK	Set record locking information.
F_SETLKW	Set record locking information; wait if blocked.

Table 6.2—File Descriptor Flags Used for *fcntl*()

Constant	Description
FD_CLOEXEC	Close the file descriptor upon execution of an exec-family function.

Table 6.3—*l_type* Values for Record Locking With *fcntl*()

Constant	Description
F_RDLCK	Shared or read lock.
F_UNLCK	Unlock.
F_WRLCK	Exclusive or write lock.

Table 6.4—*oflag* Values for *open*

Constant	Description
O_CREAT	Create file if it does not exist.
O_EXCL	Exclusive use flag.
O_NOCTTY	Do not assign a controlling terminal.
O_TRUNC	Truncate flag.

F_GETFD Get the file descriptor flags, as defined in Table 6.2, that are associated with the file descriptor *fildes*. File descriptor flags are associated with a single file descriptor and do not affect other file descriptors that refer to the same file.

F_SETFD Set the file descriptor flags, as defined in Table 6.2, that are associated with *fildes* to the third argument, *arg*, taken as type *int*. If the FD_CLOEXEC flag is zero, the file shall remain open across *exec* functions; otherwise, the file shall be closed upon successful execution of an *exec* function.

F_GETFL Get the file status flags, as defined in Table 6.5, and file access modes for the open file description associated with *fildes*. The file access modes defined in Table 6.6 can be extracted from the return value using the mask O_ACCMODE, which is defined in `<fcntl.h>`. File status flags and file access modes are associated with the open file description and do not affect other file

Table 6.5—File Status Flags Used for *open*() and *fcntl*()

Constant	Description
O_APPEND	Set append mode.
O_DSYNC	Write according to synchronized I/O data integrity completion.
O_NONBLOCK	No delay.
O_RSYNC	Synchronized read I/O operations.
O_SYNC	Write according to synchronized I/O file integrity completion.

Table 6.6—File Access Modes Used for *open*() and *fcntl*()

Constant	Description
O_RDONLY	Open for reading only.
O_RDWR	Open for reading and writing.
O_WRONLY	Open for writing only.

Table 6.7—Mask for Use With File Access Modes

Constant	Description
O_ACCMODE	Mask for file access modes.

descriptors that refer to the same file with different open file descriptions.

F_SETFL Set the file status flags, as defined in Table 6.5, for the open file description associated with *fildes* from the corresponding bits in the third argument, *arg*, taken as type *int*. Bits corresponding to the file access modes (as defined in Table 6.6) and the *oflag* values (as defined in Table 6.4) that are set in *arg* are ignored. If any bits in *arg* other than those mentioned here are changed by the application, the result is unspecified.

The following commands are available for advisory record locking. Advisory record locking shall be supported for regular files, and may be supported for other files.

F_GETLK Get the first lock that blocks the lock description pointed to by the third argument, *arg*, taken as a pointer to type *struct flock* (see below). The information retrieved overwrites the information passed to *fcntl*() in the *flock* structure. If no lock is found that would prevent this lock from being created, the structure shall be left unchanged by this function call 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* (see below). F_SETLK is used to 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 by the `<fcntl.h>` header. If a shared or exclusive lock cannot be set, *fcntl*() shall return immediately.

F_SETLKW This command is the same as 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, the *fcntl*() shall be interrupted. Upon return from the signal handler of the process, *fcntl*() shall return –1 with *errno* set to [EINTR], and the lock operation shall not be done.

The *flock* structure, defined by the `<fcntl.h>` header, describes an advisory lock. It includes the members shown in Table 6.8.

Table 6.8—*flock* Structure

Member Type	Member Name	Description
short	*l_type*	F_RDLCK, F_WRLCK, or F_UNLCK.
short	*l_whence*	Flag for starting offset.
off_t	*l_start*	Relative offset in bytes.
off_t	*l_len*	Size; if 0, then until EOF.
pid_t	*l_pid*	Process ID of the process holding the lock, returned with F_GETLK.

When a shared lock has been 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 value of *l_whence* is SEEK_SET, SEEK_CUR, or SEEK_END to indicate that the relative offset, *l_start* bytes, will 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. If *l_len* is negative, the result is undefined. 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, the value of *l_whence* shall be SEEK_SET.

Locks may start and extend beyond the current end of a file, but shall not start or 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 if *l_len* is set to zero. If the *flock struct* has *l_whence* and *l_start* that point to the beginning of the file, and *l_len* of zero, the entire 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 shall (respectively) 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 created using the *fork*() function.

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, the *fcntl*() function shall fail with an [EDEADLK] error.

If the Shared Memory Objects option is supported and 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.

6.5.2.3 Returns

Upon successful completion, the value returned shall depend on *cmd*. The various return values are shown in Table 6.9.

Table 6.9—*fcntl*() Return Values

Request	Return Value
F_DUPFD	A new file descriptor.
F_GETFD	Value of the flags defined in Table 6.2, but the return value shall not be negative.
F_SETFD	Value other than –1.
F_GETFL	Value of file status flags and access modes, but the return value shall not be 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.

Otherwise, a value of –1 shall be returned and *errno* shall be set to indicate the error.

6.5.2.4 Errors

If any of the following conditions occur, the *fcntl*() function shall return –1 and set *errno* to the corresponding value:

[EACCES] or [EAGAIN]

The argument *cmd* is F_SETLK, the type of lock (*l_type*) is a shared lock (F_RDLCK) or exclusive lock (F_WRLCK), 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 file descriptor.

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.

The argument *cmd* is F_SETLK or F_SETLKW, 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 argument *cmd* is F_SETLKW, and the function was interrupted by a signal.

[EINVAL] The argument *cmd* is F_DUPFD, and the third argument is negative or greater than or equal to {OPEN_MAX}.

The argument *cmd* is F_GETLK, F_SETLK, or F_SETLKW and the data to which *arg* points is not valid, or *fildes* refers to a file that does not support locking.

This implementation does not support synchronized I/O for this file. |

[EMFILE] The argument *cmd* is F_DUPFD and {OPEN_MAX} file descriptors are currently in use by this process, or no file descriptors greater than or equal to *arg* are available.

[ENOLCK] The argument *cmd* is F_SETLK or F_SETLKW, and satisfying the lock or unlock request would result in the number of locked regions in the system exceeding a system-imposed limit.

For each of the following conditions, if the condition is detected, the *fcntl*() function shall return −1 and set *errno* to the corresponding value:

[EDEADLK] The argument *cmd* is F_SETLKW, and a deadlock condition was detected.

6.5.2.5 Cross-References

close(), 6.3.1; *exec*, 3.1.2; *open*(), 5.3.1; `<fcntl.h>`, 6.5.1; 3.3.1.4.

6.5.3 Reposition Read/Write File Offset

Function: *lseek*()

6.5.3.1 Synopsis

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fildes, off_t offset, int whence);
```

6.5.3.2 Description

The *fildes* argument is an open file descriptor. The *lseek*() function shall set the file offset for the open file description associated with *fildes* as follows:

1) If *whence* is SEEK_SET, the offset is set to *offset* bytes.
2) If *whence* is SEEK_CUR, the offset is set to its current value plus *offset* bytes.
3) If *whence* is SEEK_END, the offset is set to the size of the file plus *offset* bytes.

The symbolic constants SEEK_SET, SEEK_CUR, and SEEK_END are defined in the header `<unistd.h>`.

Some devices are incapable of seeking. The value of the file offset associated with such a device is undefined. The behavior of the *lseek*() function on such devices is implementation defined.

The *lseek*() function shall allow 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 of data in the gap shall return bytes with the value zero until data is actually written into the gap.

The *lseek*() function shall not, by itself, extend the size of a file.

If *fildes* refers to a shared memory object, the result of the *lseek*() function is unspecified.

6.5.3.3 Returns

Upon successful completion, the function shall return the resulting offset location as measured in bytes from the beginning of the file. Otherwise, it shall return a value of ((*off_t*) −1), shall set *errno* to indicate the error, and the file offset shall remain unchanged by this function call.

6.5.3.4 Errors

If any of the following conditions occur, the *lseek*() function shall return −1 and set *errno* to the corresponding value:

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

[EINVAL] The *whence* argument is not a proper value, or the resulting file offset would be invalid.

[ESPIPE] The *fildes* argument is associated with a pipe or FIFO.

6.5.3.5 Cross-References

creat(), 5.3.2; *dup*(), 6.2.1; *fcntl*(), 6.5.2; *open*(), 5.3.1; *read*(), 6.4.1; *sigaction*(), 3.3.4; *write*(), 6.4.2; `<unistd.h>`, 2.9.

6.6 File Synchronization

The hardware characteristics upon which the implementation relies to assure that data is successfully transferred for synchronized I/O operations are implementation defined.

6.6.1 Synchronize the State of a File

Function: *fsync*()

6.6.1.1 Synopsis

```
#include <unistd.h>
int fsync(int fildes);
```

6.6.1.2 Description

If {_POSIX_FSYNC} is defined:

> The *fsync*() function can be used by the application to indicate that all data for the open file description named by *fildes* is to be transferred to the storage device associated with the file described by *fildes* in an implementation-defined manner. The *fsync*() function shall not return until the system has completed that action or until an error is detected.
> The conformance document shall include sufficient information for the user to determine whether it is possible to configure an application and installation to ensure that the data is stored with the degree of required stability for the intended use.
> If the Synchronized Input and Output option is supported, then the *fsync*() function forces all currently queued I/O operations associated with the file indicated by file descriptor *fildes* to the synchronized I/O completion state. All I/O operations are completed as defined for synchronized I/O file integrity completion.
> If both symbols {_POSIX_SYNCHRONIZED_IO} and {_POSIX_FSYNC} are defined, then the definition for {_POSIX_SYNCHRONIZED_IO} shall apply.

Otherwise:

> Either the implementation shall support the *fsync*() function as described above or the *fsync*() function shall fail.

NOTE — It is expected that a future revision will make this interface mandatory for conforming implementations and regular files. The text of the support condition will be removed at the time the contemplated revision is approved. Since the Synchronized Input and Output option will remain optional, the semantics relating to it will remain conditional.

6.6.1.3 Returns

Upon successful completion, the *fsync*() function shall return zero. Otherwise, it shall return −1 and set *errno* to indicate the error. If the *fsync*() function fails, outstanding I/O operations are not guaranteed to have been completed

6.6.1.4 Errors

If any of the following conditions occur, the *fsync*() function shall return –1 and set *errno* to the corresponding value:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINVAL]	This implementation does not support synchronized I/O for this file.
[ENOSYS]	The function *fsync*() is not supported by this implementation.

In the event that any of the queued I/O operations fail, *fsync*() shall return the error conditions defined for *read*() and *write*().

6.6.1.5 Cross-References

aio_fsync(), 6.7.9; *fdatasync*(), 6.6.2; *fcntl*(), 6.5.2; *open*(), 5.3.1; *read*(), 6.4.1; *write*(), 6.4.2.

6.6.2 Synchronize the Data of a File

Function: *fdatasync*()

6.6.2.1 Synopsis

```
#include <unistd.h>
int fdatasync(int fildes);
```

6.6.2.2 Description

If {_POSIX_SYNCHRONIZED_IO} is defined:

> The *fdatasync*() function forces all currently queued I/O operations associated with the file indicated by file descriptor *fildes* to the synchronized I/O completion state.
> The functionality is as described for *fsync*() (with the symbol {_POSIX_SYNCHRONIZED_IO} defined) with the exception that all I/O operations are completed as defined for synchronized I/O data integrity completion.

Otherwise:

> Either the implementation shall support the *fdatasync*() function as described above or the *fdatasync*() function shall fail.

6.6.2.3 Returns

If successful, the *fdatasync*() function shall return the value 0. Otherwise, the function shall return the value –1 and set *errno* to indicate the error. If the *fdatasync*() function fails, outstanding I/O operations are not guaranteed to have been completed.

6.6.2.4 Errors

If any of the following conditions occur, the *fdatasync*() function shall return –1 and set *errno* to the corresponding value:

[EBADF]	The *fildes* argument is not a valid file descriptor open for writing.
[EINVAL]	This implementation does not support synchronized I/O for this file.

[ENOSYS] The function *fdatasync*() is not supported by this implementation.

In the event that any of the queued I/O operations fail, *fdatasync*() shall return the error conditions defined for *read*() and *write*().

6.6.2.5 Cross-References

aio_fsync(), 6.7.9; *fcntl*(), 6.5.2; *fsync*(), 6.6.1; *open*(), 5.3.1; *read*(), 6.4.1; *write*(), 6.4.2.

6.7 Asynchronous Input and Output

6.7.1 Data Definitions for Asynchronous Input and Output

Inclusion of the `<aio.h>` header may make visible the symbols allowed by this part of ISO/IEC 9945 to be in the headers `<sys/types.h>`, `<signal.h>`, `<time.h>`, and `<fcntl.h>`.

6.7.1.1 Asynchronous I/O Control Block

An asynchronous I/O control block structure *aiocb* is used in many of the asynchronous I/O function interfaces. It is defined in `<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.

Implementations may add extensions as permitted in 1.3.1.1, item (2). Adding extensions to this structure, which may change the behavior of the application with respect to this standard when those fields in the structure are uninitialized, also requires that the extension be enabled as required by 1.3.1.1.

The *aio_fildes* element is the file descriptor on which the asynchronous operation is to be 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 multiprocessor or when requests of differing priorities are submitted at the same time, the ordering restriction may be relaxed; the implementation shall document under what circumstances the ordering restriction may be relaxed. 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 6.4.1 and 6.4.2, 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 shall be within the range zero through {AIO_PRIO_DELTA_MAX}, inclusive. The order of processing of requests submitted by processes whose schedulers are not SCHED_FIFO or SCHED_RR 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 shall be 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* shall have 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 3.3.1.2. 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]. Conforming 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.

6.7.1.2 Manifest Constants

The header `<aio.h>` shall define the following symbols:

AIO_CANCELED

A return value indicating that all requested operations have been canceled (see 6.7.7).

AIO_NOTCANCELED

A return value indicating that some of the requested operations could not be canceled since they are in progress (see 6.7.7).

AIO_ALLDONE	A return value indicating that none of the requested operations could be canceled since they are already complete (see 6.7.7).
LIO_WAIT	A *lio_listio*() synchronization option indicating that the calling thread is to suspend until the *lio_listio*() operation is complete.
LIO_NOWAIT	A *lio_listio*() synchronization option indicating that the calling process is to continue execution while the *lio_listio*() operation is being performed, and no notification shall be given when the operation is complete.
LIO_READ	A *lio_listio*() element operation option requesting a read.
LIO_WRITE	A *lio_listio*() element operation option requesting a write.
LIO_NOP	A *lio_listio*() element operation option indicating that no transfer is requested.

6.7.2 Asynchronous Read

Function: *aio_read*()

6.7.2.1 Synopsis

```
#include <aio.h>
int aio_read(struct aiocb *aiocbp);
```

6.7.2.2 Description

If {_POSIX_ASYNCHRONOUS_IO} is defined:

The *aio_read*() function allows the calling process to read *aiocbp->aio_nbytes* from the file associated with *aiocbp->aio_fildes* into the buffer pointed to by *aiocbp->aio_buf* (see 6.4.1). The function call returns when the read request has been initiated or queued to the file or device (even when the data cannot be delivered immediately). If {_POSIX_PRIORITIZED_IO} is defined and prioritized I/O is supported for this file, then the asynchronous operation is submitted at a priority equal to the scheduling priority of the process minus *aiocbp->aio_reqprio*. 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 returns 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 *aiocbp->aio_lio_opcode* field is 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.

If {_POSIX_SYNCHRONIZED_IO} is defined and synchronized I/0 is enabled on the file associated with *aiocbp->aio_fildes*, the behavior of this function shall be according to the definitions of synchronized I/0 data integrity completion and synchronized I/0 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.

Otherwise:

> Either the implementation shall support the *aio_read*() function as described above or the *aio_read*() function shall fail.

6.7.2.3 Returns

The *aio_read*() function shall return the value zero to the calling process if the I/O operation is successfully queued; otherwise, the function shall return the value −1 and set *errno* to indicate the error.

6.7.2.4 Errors

If any of the following conditions occur, the *aio_read*() function shall return −1 and set *errno* to the corresponding value:

[EAGAIN]	The requested asynchronous I/O operation was not queued due to system resource limitations.
[ENOSYS]	The *aio_read*() function is not supported by this implementation.

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 shall be set to −1, and the error status of the asynchronous operation shall be set to the corresponding value.

[EBADF]	The *aiocbp->aio_fildes* argument is not a valid file descriptor open for reading.
[EINVAL]	The file offset value implied by *aiocbp->aio_offset* would be invalid, *aiocbp->aio_reqprio* is not a valid value, or *aiocbp->aio_nbytes* is an invalid value.

In the case that the *aio_read*() successfully queues the I/O operation but the operation is subsequently canceled or encounters an error, the return status of the asynchronous operation shall be one of the values normally returned by the *read*() function call. In addition, the error status of the asynchronous operation shall be 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.

6.7.2.5 Cross-References

aio_write(), 6.7.3; *lio_listio*(), 6.7.4; *aio_error*(), 6.7.5; *aio_return*(), 6.7.6; *aio_cancel*(), 6.7.7; *read*(), 6.4.1; *lseek*(), 6.5.3; *close*(), 6.3.1; *_exit*(), 3.2.2; *exec*, 3.1.2; *fork*(), 3.1.1.

6.7.3 Asynchronous Write

Function: *aio_write*()

6.7.3.1 Synopsis

```
#include <aio.h>
int aio_write(struct aiocb *aiocbp);
```

6.7.3.2 Description

If {_POSIX_ASYNCHRONOUS_IO} is defined:

The *aio_write*() function allows the calling process to write *aiocbp->aio_nbytes* to the file associated with *aiocbp->aio_fildes* from the buffer pointed to by *aiocbp->aio_buf* (see 6.4.2). The function call shall return when the write request has been initiated or, at a minimum, queued to the file or device. If {_POSIX_PRIORITIZED_IO} is defined and prioritized I/O is supported for this file, then the asynchronous operation is submitted at a priority equal to the scheduling priority of the process minus *aiocbp->aio_reqprio*. The *aiocbp* 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 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. If O_APPEND is set for the file descriptor, write operations append to the file in the same order as the calls were made. After a successful call to enqueue an asynchronous I/O operation, the value of the file offset for the file is unspecified.
The *aiocbp->aio_lio_opcode* field is ignored by *aio_write*().
Simultaneous asynchronous operations using the same *aiocbp* produce undefined results.
If {_POSIX_SYNCHRONIZED_IO} is defined and 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.

Otherwise:

Either the implementation shall support the *aio_write*() function as described above or the *aio_write*() function shall fail.

6.7.3.3 Returns

The *aio_write*() function shall return the value zero to the calling process if the I/O operation is successfully queued; otherwise, the function shall return the value –1 and set *errno* to indicate the error.

6.7.3.4 Errors

If any of the following conditions occur, the *aio_write*() function shall return –1 and set *errno* to the corresponding value:

[EAGAIN]	The requested asynchronous I/O operation was not queued due to system resource limitations.
[ENOSYS]	The *aio_write*() function is not supported by this implementation.

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 shall be set to the corresponding value.

[EBADF]	The *aiocbp->aio_fildes* argument is not a valid file descriptor open for writing.

[EINVAL] The file offset value implied by *aiocbp->aio_offset* would be invalid, *aiocbp->aio_reqprio* is not a valid value, or *aiocbp->aio_nbytes* is an invalid value.

In the case that the *aio_write*() successfully queues the I/O operation, the return status of the asynchronous operation shall be one of the values normally returned by the *write*() function call. If the operation is successfully queued but is subsequently canceled or encounters an error, the error status for the asynchronous operation shall contain 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.

6.7.3.5 Cross-References

aio_read(), 6.7.2; *lio_listio*(), 6.7.4; *aio_error*(), 6.7.5; *aio_return*(), 6.7.6; *aio_cancel*(), 6.7.7; *write*(), 6.4.2; *lseek*(), 6.5.3; *close*(), 6.3.1; *_exit*(), 3.2.2; *exec*, 3.1.2; *fork*(), 3.1.1.

6.7.4 List Directed I/O

Function: *lio_listio*()

6.7.4.1 Synopsis

```
#include <aio.h>
int lio_listio(int mode, struct aiocb *const list[], int nent,
           struct sigevent *sig);
```

6.7.4.2 Description

If {_POSIX_ASYNCHRONOUS_IO} is defined:

The *lio_listio*() function allows the calling process to initiate a list of I/O requests with a single function call. The *mode* argument takes one of the values LIO_WAIT or LIO_NOWAIT declared in 6.7.1 and determines whether the function returns when the I/O operations have been completed, or as soon as the operations have been queued. If the *mode* argument is LIO WAIT, the function waits until all I/O is complete and the *sig* argument is ignored.

If the *mode* argument is LIO_NOWAIT, the function returns immediately, and signal delivery shall occur, according to the *sig* argument, when all the I/O operations complete. If the *mode* argument is LIO_NOWAIT, the function returns immediately, and asynchronous notification shall occur, according to the sig argument, when all the I/O operations complete. If *sig* is **NULL**, then no asynchronous notification occurs. If *sig* is not **NULL**, asynchronous notification shall occur as specified in 3.3.1.2 when all the requests in *list* have completed.

The I/O requests enumerated by *list* are submitted in an unspecified order.

The *list* argument is an array of pointers to *aiocb* structures. The array contains *nent* elements. The array may contain *NULL* elements, which shall be ignored.

The *aio_lio_opcode* field of each *aiocb* structure specifies the operation to be performed. The supported operations are LIO_READ, LIO_WRITE, and LIO_NOP; these symbols are defined in 6.7.1. The LIO_NOP operation causes the list entry to be ignored. If the *aio_lio_opcode* element is equal to LIO_READ, then an I/O operation is submitted as if by a call to *aio_read*() with the *aiocbp* equal to the address of the *aiocb* structure. If the *aio_lio_opcode* element is equal to LIO_WRITE, then an I/O operation is submitted as if by a call to *aio_write*() with the *aiocbp* equal to the address of the *aiocb* structure.

The *aio_fildes* member specifies the file descriptor on which the operation is to be performed.

The *aio_buf* member specifies the address of the buffer to or from which the data is to be transferred.

The *aio_nbytes* member specifies the number of bytes of data to be transferred.

The members of the *aiocb* structure further describe the I/O operation to be performed, in a manner identical to that of the corresponding *aiocb* structure when used by the *aio_read*() and *aio_write*() functions.

The *nent* argument specifies how many elements are members of the list, that is, the length of the array.

The behavior of this function is altered according to the definitions of synchronized I/O data integrity completion and synchronized I/O file integrity completion if synchronized I/O is enabled on the file associated with *aio_fildes*.

Otherwise:

Either the implementation shall support the *lio_listio*() function as described above or the *lio_listio*() function shall fail.

6.7.4.3 Returns

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* structure. The error statuses so returned are identical to those returned as the result of an *aio_read*() or *aio_write*() function.

6.7.4.4 Errors

If any of the following conditions occur, the *lio_listio*() function shall return –1 and set *errno* to the corresponding value:

[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 systemwide 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 a 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.
[ENOSYS]	The *lio_listio*() function is not supported by this implementation.

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.
[EINPROGRESS]	The requested I/O is in progress.

6.7.4.5 Cross-References

aio_read(), 6.7.2; *aio_write*(), 6.7.3; *aio_error*(), 6.7.5; *aio_return*(), 6.7.6; *aio_cancel*(), 6.7.7; *read*(), 6.4.1; *lseek*(), 6.5.3; *close*(), 6.3.1; *_exit*(), 3.2.2; *exec*, 3.1.2; *fork*(), 3.1.1.

6.7.5 Retrieve Error Status of Asynchronous I/O Operation

Function: *aio_error*()

6.7.5.1 Synopsis

```
#include <aio.h>
int aio_error (const struct aiocb *aiocbp);
```

6.7.5.2 Description

If {_POSIX_ASYNCHRONOUS_IO} is defined:

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

Otherwise:

> Either the implementation shall support the *aio_error*() function as described above or the *aio_error*() function shall fail.

6.7.5.3 Returns

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

6.7.5.4 Errors

If any of the following conditions occur, the *aio_error*() function shall return –1 and set *errno* to the corresponding value:

[ENOSYS] The *aio_error*() function is not supported by this implementation.

For each of the following conditions, if the condition is detected, the *aio_error*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] The *aiocbp* argument does not refer to an asynchronous operation whose return status has not yet been retrieved.

6.7.5.5 Cross-References

aio_read(), 6.7.2; *aio_write*(), 6.7.3; *aio_fsync*(), 6.7.9; *lio_listio*(), 6.7.4; *aio_return*(), 6.7.6; *aio_cancel*(), 6.7.7; *read*(), 6.4.1; *lseek*(), 6.5.3; *close*(), 6.3.1; *_exit*(), 3.2.2; *exec*, 3.1.2; *fork*(), 3.1.1.

6.7.6 Retrieve Return Status of Asynchronous I/O Operation

Function: *aio_return*()

6.7.6.1 Synopsis

```
#include <aio.h>
ssize_t aio_return(struct aiocb *aiocbp);
```

6.7.6.2 Description

If {_POSIX_ASYNCHRONOUS_IO} is defined:

> The *aio_return*() function returns 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.

Otherwise:

> Either the implementation shall support the *aio_return*() function as described above or the *aio_return*() function shall fail.

6.7.6.3 Returns

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.

6.7.6.4 Errors

If any of the following conditions occur, the *aio_return*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] The *aiocbp* argument does not refer to an asynchronous operation whose return status has not yet been retrieved.

[ENOSYS] The *aio_return*() function is not supported by this implementation.

6.7.6.5 Cross-References

aio_read(), 6.7.2; *aio_write*(), 6.7.3; *aio_fsync*(), 6.7.9; *lio_listio*(), 6.7.4; *aio_error*(), 6.7.5; *aio_cancel*(), 6.7.7; *read*(), 6.4.1; *lseek*(), 6.5.3; *close*(), 6.3.1; *_exit*(), 3.2.2; *exec*, 3.1.2; *fork*(), 3.1.1.

6.7.7 Cancel Asynchronous I/O Request

Function: *aio_cancel*()

6.7.7.1 Synopsis

```
#include <aio.h>
int aio_cancel (int fildes, struct aiocb *aiocbp);
```

6.7.7.2 Description

If {_POSIX_ASYNCHRONOUS_IO} is defined:

> The *aio_cancel*() function attempts to cancel one or more asynchronous I/O request 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* are canceled
> Normal signal 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 is set to [ECANCELED] and the return status is –1. For requested operations that are not successfully canceled, the *aiocbp* is not 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.

Otherwise:

> Either the implementation shall support the *aio_cancel*() function as described above or the *aio_cancel*() function shall fail.

Which operations are cancelable is implementation defined.

6.7.7.3 Returns

The *aio_cancel*() function returns the value AIO_CANCELED to the calling process if the requested operation(s) were canceled. The value AIO_NOTCANCELED is 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.

6.7.7.4 Errors

If any of the following conditions occur, the *aio_cancel*() function shall return –1 and set *errno* to the corresponding value:

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

[ENOSYS] The *aio_cancel*() function is not supported by this implementation.

6.7.7.5 Cross-References

aio_write(), 6.7.3; *aio_read*(), 6.7.2.

6.7.8 Wait for Asynchronous I/O Request

Function: *aio_suspend*()

6.7.8.1 Synopsis

```
#include <aio.h>
int aio_suspend (const struct aiocb *const list[ ], int nent,
            const struct timespec *timeout);
```

6.7.8.2 Description

If {_POSIX_ASYNCHRONOUS_IO} is defined:

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 (i.e., 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 process. 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 shall have been used in initiating an asynchronous I/O request via *aio_read*(), *aio_write*(), or *lio_listio*(). This array may contain **NULL** pointers, which shall be 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.

Otherwise:

Either the implementation shall support the *aio_suspend*() function as described above or the *aio_suspend*() function shall fail.

6.7.8.3 Returns

If the *aio_suspend*() function returns after one or more asynchronous I/0 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.

6.7.8.4 Errors

If any of the following conditions occur, the *aio_suspend*() function shall return –1 and set *errno* to the corresponding value:

[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.
[ENOSYS]	The *aio_suspend*() function is not supported by this implementation.

6.7.8.5 Cross-References

aio_write(), 6.7.3; *aio_read*(), 6.7.2; *lio_listio*(), 6.7.4.

6.7.9 Asynchronous File Synchronization

Function: *aio_fsync*()

6.7.9.1 Synopsis

```
#include <aio.h>
int aio_fsync (int op, struct aiocb *aiocbp);
```

6.7.9.2 Description

If {_POSIX_ASYNCHRONOUS_IO} and {_POSIX_SYNCHRONIZED_IO} are defined:

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

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

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

The *aiocbp* argument refers to an asynchronous I/O control block. The *aiocbp* value may be used as an argument to *aio_error*() and *aio_return*() in order to determine the error status and return status, respectively, of the asynchronous operation while it is proceeding. When the request is queued, the error status for the operation shall be [EINPROGRESS]. When all data has been successfully transferred, the error status shall be reset to reflect the success or failure of the operation. If the operation does not complete successfully, the error status for the operation shall be set to indicate the error. The *aio_sigevent* member shall determine the asynchronous notification to occur as specified in 3.3.1.2 when all operations have achieved synchronized I/O completion. All other members of the structure referenced by *aiocbp* are ignored. If the control block referenced by *aiocbp* becomes an illegal address prior to asynchronous I/O completion, then the behavior is undefined.

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

If *aiocbp* is **NULL**, then no status is returned in *aiocbp*, and no signal is generated upon completion of the operation.

Otherwise:

> Either the implementation shall support the *aio_fsync*() function as described above or the *aio_fsync*() function shall fail.

6.7.9.3 Returns

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

6.7.9.4 Errors

If any of the following conditions occur, the *aio_fsync*() function shall return –1 and set *errno* to the corresponding value:

[EAGAIN]	The requested asynchronous operation was not queued due to temporary resource limitations.
[EBADF]	The *aio_fildes* member of the *aiocb* structure referenced by the *aiocbp* argument is not a valid file descriptor open for writing.
[EINVAL]	This implementation does not support synchronized I/O for this file.
	A value of *op* other than O_DSYNC or O_SYNC was specified.
[ENOSYS]	The *aio_fsync*() function is not supported by this implementation.

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 shall be returned in the error status for the asynchronous *fsync*() operation, which can be retrieved using *aio_error*().

6.7.9.5 Cross-References

fcntl(), 6.5.2; *fdatasync*(), 6.6.2; *fsync*(), 6.6.1; *open*(), 5.3.1; *read*(), 6.4.1; *write*(), 6.4.2.

7. Device- and Class-Specific Functions

7.1 General Terminal Interface

This section describes a general terminal interface that shall be provided. It shall be supported on any asynchronous communication ports if the implementation provides them. It is implementation defined whether this interface supports network connections or synchronous ports or both. The conformance document shall describe which device types are supported by these interfaces. Certain functions in this section apply only to the controlling terminal of a process; where this is the case, it is so noted.

7.1.1 Interface Characteristics

7.1.1.1 Opening a Terminal Device File

When a terminal file is opened, it normally causes the process to wait until a connection is established. In practice, application programs seldom open these files; they are opened by special programs and become the standard input, output, and error files of an application.

As described in 5.3.1, opening a terminal device file with the O_NONBLOCK flag clear shall cause the process to block until the terminal device is ready and available. The CLOCAL flag can also affect *open*(). See 7.1.2.4.

7.1.1.2 Process Groups

A terminal may have a foreground process group associated with it. This foreground process group plays a special role in handling signal-generating input characters, as discussed below in 7.1.1.9.

If the implementation supports Job Control (if {_POSIX_JOB_CONTROL} is defined; see 2.9), command interpreter processes supporting job control can allocate the terminal to different *jobs*, or process groups, by placing related processes in a single process group and associating this process group with the terminal. The foreground process group of a terminal may be set or examined by a process, assuming the permission requirements in this section are met; see 7.2.3 and 7.2.4. The terminal interface aids in this allocation by restricting access to the terminal by processes that are not in the foreground process group; see 7.1.1.4.

When there is no longer any process whose process ID or process group ID matches the process group ID of the foreground process group, the terminal shall have no foreground process group. It is unspecified whether the terminal has a foreground process group when there is no longer any process whose process group ID matches the process group ID of the foreground process group, but there is a process whose process ID matches. No actions defined by this part of ISO/IEC 9945, other than allocation of a controlling terminal as described in 7.1.1.3 or a successful call to *tcsetpgrp*(), shall cause a process group to become the foreground process group of a terminal.

7.1.1.3 The Controlling Terminal

A terminal may belong to a process as its controlling terminal. Each process of a session that has a controlling terminal has the same controlling terminal. A terminal may be the controlling terminal for at most one session. The controlling terminal for a session is allocated by the session leader in an implementation-defined manner. If a session leader has no controlling terminal and opens a terminal device file that is not already associated with a session without using the O_NOCTTY option (see 5.3.1), it is implementation defined whether the terminal becomes the controlling terminal of the session leader. If a process that is not a session leader opens a terminal file, or the O_NOCTTY option is used on *open*(), that terminal shall not become the controlling terminal of the calling process. When a controlling terminal becomes associated with a session, its foreground process group shall be set to the process group of the session leader.

The controlling terminal is inherited by a child process during a *fork*() function call. A process relinquishes its controlling terminal when it creates a new session with the *setsid*() function; other processes remaining in the old session that had this terminal as their controlling terminal continue to have it. Upon the close of the last file descriptor in the system (whether or not it is in the current session) associated with the controlling terminal, it is unspecified whether all processes that had that terminal as their controlling terminal cease to have any controlling terminal. Whether and how a session leader can reacquire a controlling terminal after the controlling terminal has been relinquished in this fashion is unspecified. A process does not relinquish its controlling terminal simply by closing all of its file descriptors associated with the controlling terminal if other processes continue to have it open.

When a controlling process terminates, the controlling terminal is disassociated from the current session, allowing it to be acquired by a new session leader. Subsequent access to the terminal by other processes in the earlier session may be denied, with attempts to access the terminal treated as if modem disconnect had been sensed.

7.1.1.4 Terminal Access Control

If a process is in the foreground process group of its controlling terminal, read operations shall be allowed as described in 7.1.1.5. For those implementations that support Job Control, any attempts by a process in a background process group to read from its controlling terminal shall cause its process group to be sent a SIGTTIN signal unless one of the following special cases apply: If the reading process is ignoring or blocking the SIGTTIN signal, or if the process group of the reading process is orphaned, the *read*() returns −1 with *errno* set to [EIO], and no signal is sent. The default action of the SIGTTIN signal is to stop the process to which it is sent. See 3.3.1.1.

If a process is in the foreground process group of its controlling terminal, write operations shall be allowed as described in 7.1.1.8. Attempts by a process in a background process group to write to its controlling terminal shall cause the process group to be sent a SIGTTOU signal unless one of the following special cases apply: If TOSTOP is not set, or if TOSTOP is set and the process is ignoring or blocking the SIGTTOU signal, the process is allowed to write to the terminal and the SIGTTOU signal is not sent. If TOSTOP is set, and the process group of the writing process is orphaned, and the writing process is not ignoring or blocking SIGTTOU, the *write*() returns –1 with *errno* set to [EIO], and no signal is sent.

Certain calls that set terminal parameters are treated in the same fashion as write, except that TOSTOP is ignored; that is, the effect is identical to that of terminal writes when TOSTOP is set. See 7.2.

7.1.1.5 Input Processing and Reading Data

A terminal device associated with a terminal device file may operate in full-duplex mode, so that data may arrive even while output is occurring. Each terminal device file has associated with it an *input queue*, into which incoming data is stored by the system before being read by a process. The system may impose a limit, {MAX_INPUT}, on the number of bytes that may be stored in the input queue. The behavior of the system when this limit is exceeded is implementation defined.

Two general kinds of input processing are available, determined by whether the terminal device file is in canonical mode or noncanonical mode. These modes are described in 7.1.1.6 and 7.1.1.7. Additionally, input characters are processed according to the *c_iflag* (see 7.1.2.2) and *c_lflag* (see 7.1.2.5) fields. Such processing can include *echoing*, which in general means transmitting input characters immediately back to the terminal when they are received from the terminal. This is useful for terminals that can operate in full-duplex mode.

The manner in which data is provided to a process reading from a terminal device file is dependent on whether the terminal device file is in canonical or noncanonical mode.

Another dependency is whether the O_NONBLOCK flag is set by *open*() or *fcntl*(). If the O_NONBLOCK flag is clear, then the read request shall be blocked until data is available or a signal has been received. If the O_NONBLOCK flag is set, then the read request shall be completed, without blocking, in one of three ways:

1) If there is enough data available to satisfy the entire request, the *read*() shall complete successfully and return the number of bytes read.
2) If there is not enough data available to satisfy the entire request, the *read*() shall complete successfully, having read as much data as possible, and return the number of bytes it was able to read.
3) If there is no data available, the *read*() shall return –1 with *errno* set to [EAGAIN].

When data is available depends on whether the input processing mode is canonical or noncanonical. The following subclauses, 7.1.1.6 and 7.1.1.7, describe each of these input processing modes.

7.1.1.6 Canonical Mode Input Processing

In canonical mode input processing, terminal input is processed in units of lines. A line is delimited by a newline (`'\n'`) character, an end-of-file (EOF) character, or an end-of-line (EOL) character. See 7.1.1.9 for more information on EOF and EOL. This means that a read request shall not return until an entire line has been typed or a signal has been received. Also, no matter how many bytes are requested in the read call, at most one line shall be returned. It is not, however, necessary to read a whole line at once; any number of bytes, even one, may be requested in a read without losing information.

If {MAX_CANON} is defined for this terminal device, it is a limit on the number of bytes in a line. The behavior of the system when this limit is exceeded is implementation defined. If {MAX_CANON} is not defined, there is no such limit; see 2.8.5.

Erase and kill processing occur when either of two special characters, the ERASE and KILL characters (see 7.1.1.9), is received. This processing affects data in the input queue that has not yet been delimited by a newline (NL), EOF, or EOL character. This undelimited data makes up the current line. The ERASE character deletes the last character in the current line, if there is any. The KILL character deletes all data in the current line, if there is any. The ERASE and KILL characters have no effect if there is no data in the current line. The ERASE and KILL characters themselves are not placed in the input queue.

7.1.1.7 Noncanonical Mode Input Processing

In noncanonical mode input processing, input bytes are not assembled into lines, and erase and kill processing does not occur. The values of the MIN and TIME members of the *c_cc* array are used to determine how to process the bytes received.

MIN represents the minimum number of bytes that should be received when the *read*() function successfully returns. TIME is a timer of 0,1 second granularity that is used to time out short-term or bursty data transmissions. If MIN is greater than {MAX_INPUT}, the response to the request is undefined. The four possible values for MIN and TIME and their interactions are described below.

7.1.1.7.1 Case A: MIN > 0, TIME > 0

In this case TIME serves as an interbyte timer and is activated after the first byte is received. Since it is an interbyte timer, it is reset after a byte is received. The interaction between MIN and TIME is as follows: as soon as one byte is received, the interbyte timer is started. If MIN bytes are received before the interbyte timer expires (remember that the timer is reset upon receipt of each byte), the read is satisfied. If the timer expires before MIN bytes are received, the characters received to that point are returned to the user. Note that if TIME expires, at least one byte shall be returned because the timer would not have been enabled unless a byte was received. In this case (MIN > 0, TIME > 0), the read shall block until the MIN and TIME mechanisms are activated by the receipt of the first byte or until a signal is received. If data is in the buffer at the time of the *read*(), the result shall be as if data had been received immediately after the *read*().

7.1.1.7.2 Case B: MIN > 0, TIME = 0

In this case, since the value of TIME is zero, the timer plays no role and only MIN is significant. A pending read is not satisfied until MIN bytes are received (i.e., the pending read shall block until MIN bytes are received) or a signal is received. A program that uses this case to read record-based terminal I/O may block indefinitely in the read operation.

7.1.1.7.3 Case C: MIN = 0, TIME > 0

In this case, since MIN = 0, TIME no longer represents an interbyte timer. It now serves as a read timer that is activated as soon as the *read*() function is processed. A read is satisfied as soon as a single byte is received or the read timer expires. Note that in this case if the timer expires, no bytes shall be returned. If the timer does not expire, the only way the read can be satisfied is if a byte is received. In this case, the read shall not block indefinitely waiting for a byte; if no byte is received within TIME*0,1 seconds after the read is initiated, the *read*() shall return a value of zero, having read no data. If data is in the buffer at the time of the *read*(), the timer shall be started as if data had been received immediately after the *read*().

7.1.1.7.4 Case D: MIN = 0, TIME = 0

The minimum of either the number of bytes requested or the number of bytes currently available shall be returned without waiting for more bytes to be input. If no characters are available, *read*() shall return a value of zero, having read no data.

7.1.1.8 Writing Data and Output Processing

When a process writes one or more bytes to a terminal device file, they are processed according to the *c_oflag* field (see 7.1.2.3). The implementation may provide a buffering mechanism; as such, when a call to *write*() completes, all of the bytes written have been scheduled for transmission to the device, but the transmission will not necessarily have completed. See also 6.4.2 for the effects of O_NONBLOCK on *write*().

7.1.1.9 Special Characters

Certain characters have special functions on input or output or both. These functions are summarized as follows:

INTR	Special character on input and recognized if the ISIG flag (see 7.1.2.5) is enabled. It generates a SIGINT signal that is sent to all processes in the foreground process group for which the terminal is the controlling terminal. If ISIG is set, the INTR character is discarded when processed.
QUIT	Special character on input and recognized if the ISIG flag is enabled. It generates a SIGQUIT signal that is sent to all processes in the foreground process group for which the terminal is the controlling terminal. If ISIG is set, the QUIT character is discarded when processed.
ERASE	Special character on input and recognized if the ICANON flag is set. It erases the last character in the current line; see 7.1.1.6. The ERASE character shall not erase beyond the start of a line, as delimited by an NL, EOF, or EOL character. If ICANON is set, the ERASE character is discarded when processed.
KILL	Special character on input and recognized if the ICANON flag is set. It deletes the entire line, as delimited by a NL, EOF, or EOL character. If ICANON is set, the KILL character is discarded when processed.
EOF	Special character on input and recognized if the ICANON flag is set. When received, all the bytes waiting to be read are immediately passed to the process, without waiting for a newline, and the EOF is discarded. Thus, if there are no bytes waiting (that is, the EOF occurred at the beginning of a line), a byte count of zero shall be returned from the *read*(), representing an end-of-file indication. If ICANON is set, the EOF character is discarded when processed.
NL	Special character on input and recognized if the ICANON flag is set. It is the line delimiter (`'\n'`).
EOL	Special character on input and recognized if the ICANON flag is set. It is an additional line delimiter, like NL.
SUSP	Recognized on input if Job Control is supported (see 7.1.2.6). If the ISIG flag is enabled, receipt of the SUSP character causes a SIGTSTP signal to be sent to all processes in the foreground process group for which the terminal is the controlling terminal, and the SUSP character is discarded when processed.
STOP	Special character on both input and output and recognized if the IXON (output control) or IXOFF (input control) flag is set. It can be used to temporarily suspend output. It is useful with CRT terminals to prevent output from disappearing before it can be read. If IXON is set, the STOP character is discarded when processed.
START	Special character on both input and output and recognized if the IXON (output control) or IXOFF (input control) flag is set. Can be used to resume output that has been suspended by a STOP character. If IXON is set, the START character is discarded when processed.
CR	Special character on input and recognized if the ICANON flag is set; it is the `'\r'`, as denoted in the C Standard {2}. When ICANON and ICRNL are set and IGNCR is not set, this character is translated into a NL and has the same effect as a NL character.

The NL and CR characters cannot be changed. It is implementation defined whether the START and STOP characters can be changed. The values for INTR, QUIT, ERASE, KILL, EOF, EOL, and SUSP (Job Control only), shall be changeable to suit individual tastes.

If {_POSIX_VDISABLE} is in effect for the terminal file, special character functions associated with changeable special control characters can be disabled individually; see 7.1.2.6.

If two or more special characters have the same value, the function performed when that character is received is undefined.

A special character is recognized not only by its value, but also by its context; for example, an implementation may define multibyte sequences that have a meaning different from the meaning of the bytes when considered individually. Implementations may also define additional single-byte functions. These implementation-defined multibyte or single-byte functions are recognized only if the IEXTEN flag is set; otherwise, data is received without interpretation, except as required to recognize the special characters defined in this subclause (7.1.1.9).

7.1.1.10 Modem Disconnect

If a modem disconnect is detected by the terminal interface for a controlling terminal, and if CLOCAL is not set in the *c_cflag* field for the terminal (see 7.1.2.4), the SIGHUP signal is sent to the controlling process associated with the terminal. Unless other arrangements have been made, this causes the controlling process to terminate; see 3.2.2. Any subsequent call to the *read*() function shall return the value zero, indicating end of file. See 6.4.1. Thus, processes that read a terminal file and test for end-of-file can terminate appropriately after a disconnect. If the [EIO] condition specified in 6.4.1.4 that applies when the implementation supports Job Control also exists, it is unspecified whether the EOF condition or the [EIO] is returned. Any subsequent *write*() to the terminal device returns –1, with *errno* set to [EIO], until the device is closed.

7.1.1.11 Closing a Terminal Device File

The last process to close a terminal device file shall cause any output to be sent to the device and any input to be discarded. Then, if HUPCL is set in the control modes and the communications port supports a disconnect function, the terminal device shall perform a disconnect.

7.1.2 Parameters That Can Be Set

7.1.2.1 *termios* Structure

Routines that need to control certain terminal I/O characteristics shall do so by using the *termios* structure as defined in the header `<termios.h>`. The members of this structure include (but are not limited to) those shown in Table 7.1.

The types *tcflag_t* and *cc_t* shall be defined in the header `<termios.h>`. They shall be unsigned integral types.

Table 7.1—*termios* Structure

Member Type	Array Size	Member Name	Description
tcflag_t		*c_iflag*	Input modes.
tcflag_t		*c_oflag*	Output modes.
tcflag_t		*c_cflag*	Control modes.
tcflag_t		*c_lflag*	Local modes.
cc_t	NCCS	*c_cc*	Control characters.

7.1.2.2 Input Modes

Values of the *c_iflag* field, shown in Table 7.2, describe the basic terminal input control and are composed of the bitwise inclusive OR of the masks shown, which shall be bitwise distinct. The mask name symbols in this table are defined in `<termios.h>`.

Table 7.2—*termios c_iflag* Field

Mask Name	Description
BRKINT	Signal interrupt on break.
ICRNL	Map CR to NL on input.
IGNBRK	Ignore break condition.
IGNCR	Ignore CR.
IGNPAR	Ignore characters with parity errors.
INLCR	Map NL to CR on input.
INPCK	Enable input parity check.
ISTRIP	Strip character.
IXOFF	Enable start/stop input control.
IXON	Enable start/stop output control.
PARMRK	Mark parity errors.

In the context of asynchronous serial data transmission, a break condition is defined as a sequence of zero-valued bits that continues for more than the time to send one byte. The entire sequence of zero-valued bits is interpreted as a single break condition, even if it continues for a time equivalent to more than one byte. In contexts other than asynchronous serial data transmission, the definition of a break condition is implementation defined.

If IGNBRK is set, a break condition detected on input is ignored, that is, not put on the input queue and therefore not read by any process. If IGNBRK is not set and BRKINT is set, the break condition shall flush the input and output queues. If the terminal is the controlling terminal of a foreground process group, the break condition shall generate a single SIGINT signal to that foreground process group. If neither IGNBRK nor BRKINT is set, a break condition is read as a single `'\0'`, or if PARMRK is set, as `'\377'`,`'\0'`, `'\0'`.

If IGNPAR is set, a byte with a framing or parity error (other than break) is ignored.

If PARMRK is set and IGNPAR is not set, a byte with a framing or parity error (other than break) is given to the application as the three-character sequence `'\377'`, `'\0'`, *X*, where `'\377'`, `'\0'` is a two-character flag preceding each sequence and *X* is the data of the character received in error. To avoid ambiguity in this case, if ISTRIP is not set, a valid character of `'\377'` is given to the application as `'\377'`, `'\377'`. If neither PARMRK nor IGNPAR is set, a framing or parity error (other than break) is given to the application as a single character `'\0'`.

If INPCK is set, input parity checking is enabled. If INPCK is not set, input parity checking is disabled, allowing output parity generation without input parity errors. Note that whether input parity checking is enabled or disabled is independent of whether parity detection is enabled or disabled (see 7.1.2.4). If parity detection is enabled, but input parity checking is disabled, the hardware to which the terminal is connected shall recognize the parity bit, but the terminal special file shall not check whether this bit is set correctly or not.

If ISTRIP is set, valid input bytes are first stripped to seven bits; otherwise, all eight bits are processed.

If INLCR is set, a received NL character is translated into a CR character. If IGNCR is set, a received CR character is ignored (not read). If IGNCR is not set and ICRNL is set, a received CR character is translated into a NL character.

If IXON is set, start/stop output control is enabled. A received STOP character shall suspend output, and a received START character shall restart output. When IXON is set, START and STOP characters are not read, but merely perform flow control functions. When IXON is not set, the START and STOP characters are read.

If IXOFF is set, start/stop input control is enabled. The system shall transmit one or more STOP characters, which are intended to cause the terminal device to stop transmitting data, as needed to prevent the input queue from overflowing and causing the undefined behavior described in 7.1.1.5 and shall transmit one or more START characters, which are intended to cause the terminal device to resume transmitting data, as soon as the device can continue transmitting data without risk of overflowing the input queue. The precise conditions under which STOP and START characters are transmitted are implementation defined.

The initial input control value after *open*() is implementation defined.

7.1.2.3 Output Modes

Values of the *c_oflag* field describe the basic terminal output control and are composed of the bitwise inclusive OR of the following masks, which shall be bitwise distinct:

Mask Name	**Description**
OPOST	Perform output processing.

The mask name symbols for the *c_oflag* field are defined in `<termios.h>`.

If OPOST is set, output data is processed in an implementation-defined fashion so that lines of text are modified to appear appropriately on the terminal device; otherwise, characters are transmitted without change.

The initial output control value after *open*() is implementation defined.

7.1.2.4 Control Modes

Values of the *c_cflag* field, shown in Table 7.3, describe the basic terminal hardware control and are composed of the bitwise inclusive OR of the masks shown, which shall be bitwise distinct; not all values specified are required to be supported by the underlying hardware. The mask name symbols in this table are defined in `<termios.h>`.

Table 7.3—*termios c_cflag* Field

Mask Name	Description
CLOCAL	Ignore modem status lines.
CREAD	Enable receiver.
CSIZE	Number of bits per byte:
CS5	5 bits
CS6	6 bits
CS7	7 bits
CS8	8 bits
CSTOPB	Send two stop bits, else one.
HUPCL	Hang up on last close.
PARENB	Parity enable.
PARODD	Odd parity, else even.

The CSIZE bits specify the byte size in bits for both transmission and reception. This size does not include the parity bit, if any. If CSTOPB is set, two stop bits are used; otherwise, one stop bit is used. For example, at 110 baud, two stop bits are normally used.

If CREAD is set, the receiver is enabled; otherwise, no characters shall be received.

If PARENB is set, parity ,generation and detection is enabled and a parity bit is added to each character. If parity is enabled, PARODD specifies odd parity if set; otherwise, even parity is used.

If HUPCL is set, the modem control lines for the port shall be lowered when the last process with the port open closes the port or the process terminates. The modem connection shall be broken.

If CLOCAL is set, a connection does not depend on the state of the modem status lines. If CLOCAL is clear, the modem status lines shall be monitored.

Under normal circumstances, a call to the *open*() function shall wait for the modem connection to complete. However, if the O_NONBLOCK flag is set (see 5.3.1) or if CLOCAL has been set, the *open*() function shall return immediately without waiting for the connection.

If the object for which the control modes are set is not an asynchronous serial connection, some of the modes may be ignored; for example, if an attempt is made to set the baud rate on a network connection to a terminal on another host, the baud rate may or may not be set on the connection between that terminal and the machine to which it is directly connected.

The initial hardware control value after *open*() is implementation defined.

7.1.2.5 Local Modes

Values of the *c_lflag* field, shown in Table 7.4, describe the control of various functions and are composed of the bitwise inclusive OR of the masks shown, which shall be bitwise distinct. The mask name symbols in this table are defined in `<termios.h>`.

Table 7.4—*termios c_lflag* Field

Mask Name	Description
ECHO	Enable echo.
ECHOE	Echo ERASE as an error-correcting backspace.
ECHOK	Echo KILL.
ECHONL	Echo `'\n'`.
ICANON	Canonical input (erase and kill processing).
IEXTEN	Enable extended (implementation-defined) functions
ISIG	Enable signals.
NOFLSH	Disable flush after interrupt, quit, or suspend.
TOSTOP	Send SIGTTOU for background output.

If ECHO is set, input characters are echoed back to the terminal. If ECHO is not set, input characters are not echoed.

If ECHOE and ICANON are set, the ERASE character shall cause the terminal to erase the last character in the current line from the display, if possible. If there is no character to erase, an implementation may echo an indication that this was the case or do nothing.

If ECHOK and ICANON are set, the KILL character shall either cause the terminal to erase the line from the display or shall echo the `'\n'` character after the KILL character.

If ECHONL and ICANON are set, the `'\n'` character shall be echoed even if ECHO is not set.

If ICANON is set, canonical processing is enabled. This enables the erase and kill edit functions and the assembly of input characters into lines delimited by NL, EOF, and EOL, as described in 7.1.1.6.

If ICANON is not set, read requests are satisfied directly from the input queue. A read shall not be satisfied until at least MIN bytes have been received or the timeout value TIME has expired between bytes. The time value represents tenths or seconds. See 7.1.1.7 for more details.

If ISIG is set, each input character is checked against the special control characters INTR, QUIT, and SUSP (Job Control only). If an input character matches one of these control characters, the function associated with that character is performed. If ISIG is not set, no checking is done. Thus, these special input functions are possible only if ISIG is set.

If IEXTEN is set, implementation-defined functions shall be recognized from the input data. It is implementation defined how IEXTEN being set interacts with ICANON, ISIG, IXON, or IXOFF. If IEXTEN is not set, then implementation-defined functions shall not be recognized, and the corresponding input characters shall be processed as described for ICANON, ISIG, IXON, and IXOFF.

If NOFLSH is set, the normal flush of the input and output queues associated with the INTR, QUIT, and SUSP (Job Control only). characters shall not be done.

If TOSTOP is set and the implementation supports Job Control, the signal SIGTTOU is sent to the process group of a process that tries to write to its controlling terminal if it is not ill the foreground process group for that terminal. This signal, by default, stops the members of the process group. Otherwise, the output generated by that process is output to the current output stream. Processes that are blocking or ignoring SIGTTOU signals are excepted and allowed to produce output, and the SIGTTOU signal is not sent.

The initial local control value after *open*() is implementation defined.

7.1.2.6 Special Control Characters

The special control characters values are defined by the array *c_cc*. The subscript name and description for each element in both canonical and noncanonical modes are shown in Table 7.5. The subscript name symbols in this table are defined in `<termios.h>`.

Table 7.5—*termios c_cc* Special Control Characters

Subscript Usage		
Canonical Mode	**Noncanonical Mode**	**Description**
VEOF		EOF character
VEOL		EOL character
VERASE		ERASE character
VINTR	VINTR	INTR character
VKILL		KILL character
	VMIN	MIN value
VQUIT	VQUIT	QUIT character
VSUSP	VSUSP	SUSP character
	VTIME	TIME value
VSTART	VSTART	START character
VSTOP	VSTOP	STOP character

The subscript values shall be unique, except that the VMIN and VTIME subscripts may have the same values as the VEOF and VEOL subscripts, respectively.

Implementations that do not support Job Control may ignore the SUSP character value in the *c_cc* array indexed by the VSUSP subscript.

The value of NCCS (the number of elements in the *c_cc* array) is unspecified by this part of ISO/IEC 9945.

Implementations that do not support changing the START and STOP characters may ignore the character values in the *c_cc* array indexed by the VSTART and VSTOP subscripts when *tcsetattr*() is called, but shall return the value in use when *tcgetattr*() is called.

If {_POSIX_VDISABLE} is defined for the terminal device file, and the value of one of the changeable special control characters (see 7.1.1.9) is {_POSIX_VDISABLE}, that function shall be disabled, that is, no input data shall be recognized as the disabled special character. If ICANON is not set, the value of {_POSIX_VDISABLE} has no special meaning for the VMIN and VTIME entries of the *c_cc* array.

The initial values of all control characters are implementation defined.

7.1.2.7 Baud Rate Values

The baud rate values specified in Table 7.6 can be set into the *termios* structure by the baud rate functions in 7.1.3.

Table 7.6—*termios* Baud Rate Values

Name	Description		Name	Description
B0	Hang up		B600	600 baud
B50	50 baud		B1200	1200 baud
B75	75 baud		B1800	1800 baud
B110	110 baud		B2400	2400 baud
B134	134.5 baud		B4800	4800 baud
B150	150 baud		B9600	9600 baud
B200	200 baud		B19200	19200 baud
B300	300 baud		B38400	38400 baud

7.1.3 Baud Rate Functions

Functions: *cfgetispeed*(), *cfgetospeed*(), *cfsetispeed*(), *cfsetospeed*()

7.1.3.1 Synopsis

```
#include <termios.h>
speed_t cfgetospeed(const struct termios *termios_p);
int cfsetospeed(struct termios *termios_p, speed_t speed);
speed_t cfgetispeed(const struct termios *termios_p);
int cfsetispeed(struct termios *termios_p, speed_t speed);
```

7.1.3.2 Description

The following interfaces are provided for getting and setting the values of the input and output baud rates in the *termios* structure. The effects on the terminal device described below do not become effective until the *tcsetattr*() function is successfully called, and not all errors are detected until *tcsetattr*() is called as well.

The input and output baud rates are represented in the *termios* structure. The values shown in Table 7.6 are defined. The name symbols in this table are defined in `<termios.h>`.

The type *speed_t* shall be defined in `<termios.h>` and shall be an unsigned integral type.

The *termios_p* argument is a pointer to a *termios* structure.

The *cfgetospeed*() function shall return the output baud rate stored in the *termios* structure to which *termios_p* points.

The *cfgetispeed*() function shall return the input baud rate stored in the *termios* structure to which *termios_p* points.

The *cfsetospeed*() function shall set the output baud rate stored in the *termios* structure to which *termios_p* points.

The *cfsetispeed*() function shall set the input baud rate stored in the *termios* structure to which *termios_p* points.

Certain values for speeds that are set in the *termios* structure and passed to *tcsetattr*() have special meanings. These are discussed under *tcsetattr*().

The *cfgetispeed*() and *cfgetospeed*() functions return exactly the value found in the *termios* data structure, without interpretation.

Both *cfsetispeed*() and *cfsetospeed*() return a value of zero if successful and −1 to indicate an error. It is unspecified whether these return an error if an unsupported baud rate is set.

7.1.3.3 Returns

See 7.1.3.2.

7.1.3.4 Errors

This part of ISO/IEC 9945 does not specify any error conditions that are required to be detected for the *cfgetispeed*(), *cfgetospeed*(), *cfsetispeed*(), or *cfsetospeed*() functions. Some errors may be detected under conditions that are unspecified by this part of ISO/IEC 9945.

7.1.3.5 Cross-References

tcsetattr(), 7.2.1.

7.2 General Terminal Interface Control Functions

The functions that are used to control the general terminal function are described in this clause. If the implementation supports Job Control, unless otherwise noted for a specific command, these functions are restricted from use by background processes. Attempts to perform these operations shall cause the process group to be sent a SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process is allowed to perform the operation and the SIGTTOU signal is not sent.

In all the functions, *fildes* is an open file descriptor. However, the functions affect the underlying terminal file, not just the open file description associated with the file descriptor.

7.2.1 Get and Set State

Functions: *tcgetattr*(), *tcsetattr*()

7.2.1.1 Synopsis

```
#include <termios.h>
int tcgetattr(int fildes, struct termios *termios_p);
int tcsetattr(int fildes, int optional_actions,
            const struct termios *termios_p);
```

7.2.1.2 Description

The *tcgetattr*() function shall get the parameters associated with the object referred to by *fildes* and store them in the *termios* structure referenced by *termios_p*. This function is allowed from a background process; however, the terminal attributes may be subsequently changed by a foreground process. If the terminal device supports different input and output baud rates, the baud rates stored in the *termios* structure returned by *tcgetattr*() shall reflect the actual baud rates, even if they are equal. If differing baud rates are not supported, the rate returned as the output baud rate shall be the actual baud rate. The rate returned as the input baud rate shall be either the number zero or the output rate (as one of the symbolic values). Permitting either behavior is obsolescent.[3]

[3]In a future revision of this part of ISO/IEC 9945, a returned value of zero as the input baud rate when differing baud rates are not supported may no longer be permitted.

The *tcsetattr*() function shall set the parameters associated with the terminal (unless support is required from the underlying hardware that is not available) from the *termios* structure referenced by *termios_p* as follows:

1) If *optional_actions* is TCSANOW, the change shall occur immediately.
2) If *optional_actions* is TCSADRAIN, the change shall occur after all output written to *fildes* has been transmitted. This function should be used when changing parameters that affect output.
3) If *optional_actions* is TCSAFLUSH, the change shall occur after all output written to the object referred to by *fildes* has been transmitted, and all input that has been received, but not read, shall be discarded before the change is made.

The symbolic constants for the values of *optional_actions* are defined in `<termios.h>`.

The zero baud rate, B0, is used to terminate the connection. If B0 is specified as the output baud rate when *tcsetattr*() is called, the modem control lines shall no longer be asserted. Normally, this will disconnect the line.

If the input baud rate is equal to the numeral zero in the *termios* structure when *tcsetattr*() is called, the input baud rate will be changed by *tcsetattr*() to the same value as that specified by the value of the output baud rate, exactly as if the input rate had been set to the output rate by *cfsetispeed*(). This usage of zero is obsolescent.

The *tcsetattr*() function shall return success 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 does support as requested and leave all the attributes not supported by the hardware 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 is 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 in the *termios* structure whether or not it actually accepts them.

The *termios* structure may have additional fields not defined by this part of ISO/IEC 9945. The effect of the *tcsetattr*() function 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*; a Strictly Conforming POSIX.1 Application shall modify only fields and flags defined by this part of ISO/IEC 9945 between the call to *tcgetattr*() and *tcsetattr*(), leaving all other fields and flags unmodified.

No actions defined by this part of ISO/IEC 9945, 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 part of ISO/IEC 9945 to change.

7.2.1.3 Returns

Upon successful completion, a value of zero is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

7.2.1.4 Errors

If any of the following conditions occur, the *tcgetattr*() function shall return −1 and set *errno* to the corresponding value:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[ENOTTY]	The file associated with *fildes* is not a terminal.

If any of the following conditions occur, the *tcsetattr*() function shall return −1 and set *errno* to the corresponding value:

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

[EINTR]	A signal interrupted the *tcsetattr*() function.
[EINVAL]	The *optional_actions* argument is not a proper 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.

7.2.1.5 Cross-References

`<termios.h>`, 7.1.2.

7.2.2 Line Control Functions

Functions: *tcsendbreak*(), *tcdrain*(), *tcflush*(), *tcflow*()

7.2.2.1 Synopsis

```
#include <termios.h>
int tcsendbreak(int fildes, int duration);
int tcdrain(int fildes);
int tcflush(int fildes, int queue_selector);
int tcflow(int fildes, int action);
```

7.2.2.2 Description

If the terminal is using asynchronous serial data transmission, the *tcsendbreak*() function shall cause transmission of a continuous stream of zero-valued bits for a specific duration. If *duration* is zero, it shall cause transmission of zero-valued bits for at least 0,25 seconds and not more that 0,5 seconds. If *duration* is not zero, it shall send zero-valued bits for an implementation-defined period of time.

If the terminal is not using asynchronous serial data transmission, it is implementation defined whether the *tcsendbreak*() function sends data to generate a break condition (as defined by the implementation) or returns without taking any action.

The *tcdrain*() function shall wait until all output written to the object referred to by *fildes* has been transmitted.

Upon successful completion, the *tcflush*() function shall have discarded any data written to the object referred to by *fildes* but not transmitted, or data received, but not read, depending on the value of *queue_selector*:

1) If *queue_selector* is TCIFLUSH, it shall flush data received, but not read.
2) If *queue_selector* is TCOFLUSH, it shall flush data written, but not transmitted.
3) If *queue_selector* is TCIOFLUSH, it shall flush both data received but not read and data written but not transmitted.

The *tcflow*() function shall suspend transmission or reception of data on the object referred to by *fildes*, depending on the value of *action*:

1) If *action* is TCOOFF, it shall suspend output.
2) If *action* is TCOON, it shall restart suspended output.
3) 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. (See the description of IXOFF in 7.1.2.2.)
4) 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. (See the description of IXOFF in 7.1.2.2.)

The symbolic constants for the values of *queue_selector* and *action* are defined in `<termios.h>`.

The default on the opening of a terminal file is that neither its input nor its output is suspended.

7.2.2.3 Returns

Upon successful completion, a value of zero is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

7.2.2.4 Errors

If any of the following conditions occur, the *tcsendbreak*() function shall return −1 and set *errno* to the corresponding value:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[ENOTTY]	The file associated with *fildes* is not a terminal.

If any of the following conditions occur, the *tcdrain*() function shall return −1 and set *errno* to the corresponding value:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINTR]	A signal interrupted the *tcdrain*() function.
[ENOTTY]	The file associated with *fildes* is not a terminal.

If any of the following conditions occur, the *tcflush*() function shall return −1 and set *errno* to the corresponding value:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINVAL]	The *queue_selector* argument is not a proper value.
[ENOTTY]	The file associated with *fildes* is not a terminal.

If any of the following conditions occur, the *tcflow*() function shall return −1 and set *errno* to the corresponding value:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINVAL]	The *action* argument is not a proper value.
[ENOTTY]	The file associated with *fildes* is not a terminal.

7.2.2.5 Cross-References

`<termios.h>`, 7.1.2.

7.2.3 Get Foreground Process Group ID

Function: *tcgetpgrp*()

7.2.3.1 Synopsis

```
#include <sys/types.h>
pid_t tcgetpgrp (int fildes);
```

7.2.3.2 Description

If {_POSIX_JOB_CONTROL} is defined:

1) The *tcgetpgrp*() function shall return the value of the process group ID of the foreground process group associated with the terminal.
2) 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.

Otherwise:

The implementation shall either support the *tcgetpgrp*() function as described above or the *tcgetpgrp*() call shall fail.

7.2.3.3 Returns

Upon successful completion, *tcgetpgrp*() returns 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. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

7.2.3.4 Errors

If any of the following conditions occur, the *tcgetpgrp*() function shall return –1 and set *errno* to the corresponding value:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[ENOSYS]	The *tcgetpgrp*() function is not supported in this implementation.
[ENOTTY]	The calling process does not have a controlling terminal, or the file is not the controlling terminal.

7.2.3.5 Cross-References

setsid(), 4.3.2; *setpgid*(), 4.3.3; *tcsetpgrp*(), 7.2.4.

7.2.4 Set Foreground Process Group ID

Function: *tcsetpgrp*()

7.2.4.1 Synopsis

```
#include <sys/types.h>
int tcsetpgrp (int fildes, pid_t pgrp_id);
```

7.2.4.2 Description

If {_POSIX_JOB_CONTROL} is defined:

If the process has a controlling terminal, the *tcsetpgrp*() function shall set the foreground process group ID associated with the terminal to *pgrp_id*. The file associated with *fildes* must be the controlling terminal of the calling process, and the controlling terminal must be currently associated with the session of the calling process. The value of *pgrp_id* must match a process group ID of a process in the same session as the calling process.

Otherwise:

The implementation shall either support the *tcsetpgrp*() function as described above, or the *tcsetpgrp*() call shall fail.

7.2.4.3 Returns

Upon successful completion, *tcsetpgrp*() returns a value of zero. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

7.2.4.4 Errors

If any of the following conditions occur, the *tcsetpgrp*() function shall return –1 and set *errno* to the corresponding value:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINVAL]	The value of the *pgrp_id* argument is not supported by the implementation.
[ENOSYS]	The *tcsetpgrp*() function is not supported in this implementation.
[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 *pgrp_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.

8. Language-Specific Services for the C Programming Language

8.1 Referenced C Language Routines

The functions listed below are described in the indicated sections of the C Standard {2}. POSIX.1 with the C Language Binding comprises these functions, the extensions to them described in this clause, and the rest of the requirements stipulated in this part of ISO/IEC 9945. The functions appended with plus signs (+) have requirements beyond those set forth in the C Standard {2}. Any implementation claiming conformance to POSIX.1 with the C Language Binding shall comply with the requirements outlined in this clause, the requirements stipulated in the rest of this part of ISO/IEC 9945, and the requirements in the indicated sections of the C Standard {2}.

For requirements concerning conformance to this clause, see 1.3.3 and its subclauses.

4.2 Diagnostics

Functions: assert.

4.3 Character Handling

Functions: isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, tolower, toupper.

4.4 Localization

Functions: setlocale+.

4.5 Mathematics

Functions: acos, asin, atan, atan2, cos, sin, tan, cosh, sinh, tanh, exp, frexp, ldexp, log, log10, modf, pow, sqrt, ceil, fabs, floor, fmod.

4.6 Non-Local Jumps

Functions: setjm+, longimp+.

4.9 Input/Output

Functions: clearerr, fclose, feof, ferror, fflush, fgetc, fgets, fopen, fputc, fputs, fread, freopen, fseek, ftell, fwrite, getc, getchar, gets, perror, printf, fprintf, sprintf, putc, putchar, puts, remove, rename+, rewind, scanf, fscanf, sscanf, setbuf, tmpfile, tmpnam, ungetc.

4.10 General Utilities

Functions: abs, atof, atoi, atol, rand, srand, calloc, free, malloc, realloc, abort+, exit, getenv+, bsearch, qsort.

4.11 String Handling

Functions: strcpy, strncpy, strcat, strncat, strcmp, strncmp, strchr, strcspn, strpbrk, strrchr, strspn, strstr, strtok, strlen.

4.12 Date and Time

Functions: time, asctime, ctime+, gmtime+, localtime+, mktime+, strftime+.

Systems conforming to this part of ISO/IEC 9945 shall make no distinction between the “text streams” and the “binary streams” described in the C Standard {2}.

For the *fseek*() function, if the specified position is beyond end-of-file, the consequences described in *lseek*() (see 6.5.3) shall occur.

The EXIT_SUCCESS macro, as used by the *exit*() function, shall evaluate to a value of zero. Similarly, the EXIT_FAILURE macro shall evaluate to a nonzero value.

The relationship between a time in seconds since the Epoch used as an argument to *gmtime*() and the *tm* structure (defined in `<time.h>`) is that the result shall be as specified in the expression given in the definition of *seconds since the Epoch* in 2.2.2.113, where the names in the structure and in the expression correspond. If the time zone `UCT0` is in effect, this shall also be true for *localtime*() and *mktime*().

The effect of the *raise*() function shall be equivalent to calling `pthread_kill (pthread_self() ,` *sig*`)`.

8.1.1 Extensions to Time Functions

The contents of the environment variable named **TZ** (see 2.6) shall be used by the functions *ctime*(), *localtime*(), *strftime*(), and *mktime*() to override the default time zone. The value of **TZ** has one of the two forms (spaces inserted for clarity):

`:`*characters*

or:

std offset dst offset`,` *rule*

If **TZ** is of the first format (i.e., if the first character is a colon), the characters following the colon are handled in an implementation-defined manner.

The expanded format (for all **TZ**s whose value does not have a colon as the first character) is as follows:

stdoffset[*dst*[*offset*][, *start*[*/time*], *end*[*/time*]]]

Where:

std and *dst* Indicates no less than three, nor more than {TZNAME_MAX}, bytes that are the designation for the standard (*std*) or summer (*dst*) time zone. Only *std* is required; if *dst* is missing, then summer time does not apply in this locale. Upper- and lowercase letters are explicitly allowed. Any characters except a leading colon (:) or digits, the comma (,), the minus (–), the plus (+), and the null character are permitted to appear in these fields, but their meaning is unspecified.

offset Indicates the value one must add to the local time to arrive at Coordinated Universal Time. The *offset* has the form:

hh [: *mm*[: *ss*]]

The minutes (*mm*) and seconds (*ss*) are optional. The hour (*hh*) shall be required and may be a single digit. The *offset* following *std* shall be required. If no *offset* follows *dst*, summer time is assumed to be one hour ahead of standard time. One or more digits may be used; the value is always interpreted as a decimal number. The hour shall be between zero and 24, and the minutes (and seconds)—if present—between zero and 59. Use of values outside these ranges causes undefined behavior. If preceded by a "–", the time zone shall be east of the Prime Meridian; otherwise it shall be west (which may be indicated by an optional preceding "+").

rule Indicates when to change to and back from summer time. The *rule* has the form:

date / *time*`,`*date*/ *time*

where the first *date* describes when the change from standard to summer time occurs and the second *date* describes when the change back happens. Each *time* field describes when, in current local time, the change to the other time is made.

The format of *date* shall be one of the following:

J*n* The Julian day *n* ($1 \leq n \leq 365$). Leap days shall not be counted. That is, in all years—including leap years—February 28 is day 59 and March 1 is day 60. It is impossible to explicitly refer to the occasional February 29.

n The zero-based Julian day ($0 \leq n \leq 365$). Leap days shall be counted, and it is possible to refer to February 29.

M*m.n.d*

The d^{th} day ($0 \leq \delta \leq 6$) of week *n* of month *m* of the year ($1 \leq n \leq 5$, $1 \leq m \leq 12$, where week 5 means "the last *d* day in month *m*" which may occur in either the fourth or the fifth week). Week 1 is the first week in which the *d*'th day occurs. Day zero is Sunday.

The *time* has the same format as *offset* except that no leading sign ("–" or "+") shall be allowed. The default, if *time* is not given, shall be 02:00:00.

Whenever *ctime*(), *strftime*(), *mktime*(), or *localtime*() is called, the time zone names contained in the external variable *tzname* shall be set as if the *tzset*() function had been called.

Applications are explicitly allowed to change **TZ** and have the changed **TZ** apply to themselves.

8.1.2 Extensions to *setlocale*() Function

Function: *setlocale*()

8.1.2.1 Synopsis

```
#include <locale.h>
char *setlocale (int category, const char *locale) ;
```

8.1.2.2 Description

The *setlocale*() function sets, changes, or queries the locale of the process according to the values of the *category* and the *locale* arguments. The possible values for *category* include:

LC_CTYPE
LC_COLLATE
LC_TIME
LC_NUMERIC
LC_MONETARY
Implementation-defined additional categories

For POSIX.1 systems, environment variables are defined that correspond to the named categories above and that have the same spelling.

The value LC_ALL for *category* names all of the categories of the locale of the process; LC_ALL is a special constant, not a category. There is an environment variable **LC_ALL** with the semantics noted below.

The *locale* argument is a pointer to a character string that can be an explicit string, a **NULL** pointer, or a null string.

When *locale* is an explicit string, the contents of the string are implementation defined except for the value “C” The value “C” for *locale* specifies the minimal environment for C-language translation. If *setlocale*() is not invoked, the “C” locale shall be the locale of the process. The locale name “POSIX” shall be recognized. It shall provide the same semantics as the C locale for those functions defined within this part of ISO/IEC 9945 or by the C Standard {2}. Extensions or refinements to the POSIX locale beyond those provided by the C locale may be included in future revisions, and other parts of ISO/IEC 9945 are expected to add to the requirements of the POSIX locale.

When *locale* is a **NULL** pointer the locale of the process is queried according to the value of *category*. The content of the string returned is unspecified.

When *locale* is a null string, the *setlocale*() function takes the name of the new locale for the specified category from the environment as determined by the first condition met below:

1) If **LC_ALL** is defined in the environment and is not null, the value of **LC_ALL** is used.
2) If there is a variable defined in the environment with the same name as the category and that is not null, the value specified by that environment variable is used.
3) If **LANG** is defined in the environment and is not null, the value of **LANG** is used.

If the resulting value is a supported locale, *setlocale*() sets the specified category of the locale of the process to that value and returns the value specified below. If the value does not name a supported locale (and is not null), *setlocale*() returns a **NULL** pointer, and the locale of the process is not changed by this function call. If no nonnull environment variable is present to supply a value, it is implementation defined whether *setlocale*() sets the specified category of the locale of the process to a systemwide default value or to “C” or to “POSIX”. The possible actual values of the environment variables are implementation defined and should appear in the system documentation.

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*() first verifies that the values of all the environment variables it needs according to the precedence above indicate supported locales. If the value of any of these environment-variable searches yields a locale that is not supported (and nonnull), the *setlocale*() function returns a **NULL** pointer and the locale of the process is not changed. If all environment variables name supported locales, *setlocale*() then proceeds 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.

8.1.2.3 Returns

A successful call to *setlocale*() returns a string that corresponds to the locale set. The string returned is such that "a subsequent call with that string and its associated category will restore that part of the process's locale" (C Standard {2}). The string returned shall not be modified by the process, but may be overwritten by a subsequent call to the *setlocale*() function. This string is not required to be the value of the environment variable used, if one was used.

8.2 C Language Input/Output Functions

This clause describes input/output functions of the C Standard {2} and their interactions with other functions defined by this part of ISO/IEC 9945.

All functions specified in the C Standard {2} as operating on a *file name* shall operate on a *pathname*. All functions specified in the C Standard {2} as creating a file shall do so as if they called the *creat*() function with a value appropriate to the C language function for the *path* argument and a value of

```
    S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH
```

for the *mode* argument.

The type *FILE* and the terms *file position indicator* and *stream* are those defined by the C Standard {2}.

A stream is considered local to a single process. After a *fork*() call, each of the parent and child have distinct streams that share an open file description.

8.2.1 Map a Stream Pointer to a File Descriptor

Function: *fileno*()

8.2.1.1 Synopsis

```
#include <stdio.h>
int fileno(FILE *stream);
```

8.2.1.2 Description

The *fileno*() function returns the integer file descriptor associated with the *stream* (see 5.3.1).

The following symbolic values in the `<unistd.h>` header (see 2.9) define the file descriptors that shall be associated with the C language *stdin*, *stdout*, and *stderr* when the application is started:

Name	Description	Value
STDIN_FILENO	Standard input value, *stdin*.	0
STDOUT_FILENO	Standard output value, *stdout*.	1
STDERR_FILENO	Standard error value, *stderr*.	2

At entry to *main*(), these streams shall be in the same state as if they had just been opened with *fdopen*() called with a mode consistent with that required by the C Standard {2} and the file descriptor described above.

8.2.1.3 Returns

See 8.2.1.2. If an error occurs, a value of –1 is returned and *errno* is set to indicate the error.

8.2.1.4 Errors

This part of ISO/IEC 9945 does not specify any error conditions that are required to be detected for the *fileno*() function. Some errors may be detected under conditions that are unspecified by this part of ISO/IEC 9945.

8.2.1.5 Cross-References

open(), 5.3.1.

8.2.2 Open a Stream on a File Descriptor

Function: *fdopen*()

8.2.2.1 Synopsis

```
#include <stdio.h>
FILE *fdopen(int fildes, const char *type);
```

8.2.2.2 Description

The *fdopen*() routine associates a stream with a file descriptor.

The *type* argument is a character string having one of the following values:

`"r"`	Open for reading.
`"w"`	Open for writing.
`"a"`	Open for writing at end-of-file.
`"r+"`	Open for update (reading and writing).
`"w+"`	Open for update (reading and writing).
`"a+"`	Open for update (reading and writing) at end-of-file.

The meaning of these flags is exactly as specified by the C Standard {2} for *fopen*(), except that `"w"` and `"w+"` do not cause truncation of the file. Additional values for the *type* argument may be defined by an implementation.

The application shall ensure that the mode of the stream is allowed by the mode of the open file.

The file position indicator associated with the new stream is set to the position indicated by the file offset associated with the file descriptor. The error indicator and end-of-file indicator for the stream shall be cleared.

If *fildes* refers to a shared memory object, the result of the *fdopen*() function is unspecified.

8.2.2.3 Returns

If successful, the *fdopen*() function returns a pointer to a stream. Otherwise, a **NULL** pointer is returned and *errno* is set to indicate the error.

8.2.2.4 Errors

This part of ISO/IEC 9945 does not specify any error conditions that are required to be detected for the *fdopen*() function. Some errors may be detected under conditions that are unspecified by this part of ISO/IEC 9945.

8.2.2.5 Cross-References

open(), 5.3.1; *fopen*() [C Standard {2}].

8.2.3 Interactions of Other *FILE*-Type C Functions

A single open file description can be accessed both through streams and through file descriptors. Either a file descriptor or a stream will be 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 user action without affecting the underlying open file description. Some of the ways to create them include *fcntl*(), *dup*(), *fdopen*(), *fileno*(), and *fork*() (which duplicates existing ones into new processes). They can be destroyed by at least *fclose*(), *close*(), and the *exec* functions (which close some file descriptors and destroy streams).

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 in this discussion, but could give rise to one [as a consequence of *fdopen*(), *dup*(), or *fork*(), for example]. This exception does include the file descriptor underlying a stream, whether created with *fopen*() or *fdopen*(), as 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*) are defined elsewhere in this part of ISO/IEC 9945, but if two or more handles are used, and any one of them is a stream, their actions shall be coordinated as described below. If this is not done, the result is undefined.

A handle that 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 for this discussion, 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*() or *abort*(). A file descriptor is closed by *close*(), *_exit*(), or by one of the *exec* functions when FD_CLOEXEC is set on that file descriptor.

For a handle to become the active handle, the actions below must be performed between the last other use of the first handle (the current active handle) and the first other 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 handle. (If a stream function has as an underlying function that affects the file offset, the stream function will be considered to affect the file offset. The underlying functions are described below.)

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 assure that, if both handles, will ever be accessed, that they will both be 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.]

1) For the first handle, the first applicable condition below shall apply. After the actions required below are taken, the handle may be closed if it is still open.
   a) If it is a file descriptor, no action is required.
   b) If the only further action to be performed on any handle to this open file description is to close it, no action need be taken.
   c) If it is a stream that is unbuffered, no action need be taken.
   d) If it is a stream that is line-buffered and the last character 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.
   e) If it is a stream that is open for writing or append (but not also open for reading), either an *fflush*() shall occur or the stream shall be closed.
   f) If the stream is open for reading and it is at the end of the file [*feof*() is true], no action need be taken.
   g) 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, either an *fflush*() shall occur or the stream shall be closed.
   h) Otherwise, the result is undefined.
2) For the second handle: if any previous active handle has called a function that explicitly changed the file offset, except as required above for the first handle, the application shall perform an *lseek*() or an *fseek*() (as appropriate to the type of the handle) to an appropriate location.
3) 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, for example, during a *fork*() or an *_exit*().
4) The *exec* functions shall be considered to make inaccessible all streams that are open at the time they are called, independent of what streams or file descriptors may be available to the new process image.
5) Implementations shall assure that an application, even one consisting of several processes, shall yield correct results (no data is lost or duplicated when writing, all data is written in order, except as requested by seeks) when the rules above are followed, regardless of the sequence of handles used. If the rules above are not followed, the result is unspecified. When these rules are followed, it is implementation defined whether, and under what conditions, all input is seen exactly once.
6) 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.
7) Also, in the subclauses below, additional requirements on the standard I/O routines, beyond those in the C Standard {2}, are given.

8.2.3.1 *fopen*()

The *fopen*() function shall allocate a file descriptor as *open*() does.

The underlying function is *open*().

8.2.3.2 *fclose*()

The *fclose*() function shall perform a *close*() on the file descriptor that is associated with the *FILE* stream. It shall also mark for update the *st_ctime* and *st_mtime* fields of the underlying file, if the stream was writable, and if buffered data had not been written to the file yet.

The underlying functions are *write*() and *close*().

8.2.3.3 *freopen*()

The *freopen*() function has the properties of both *fclose*() and *fopen*().

8.2.3.4 ***fflush*()**

The *fflush*() function shall mark for update the *st_ctime* and *st_mtime* fields of the underlying file if the stream was writable and if buffered data had not been written to the file yet.

The underlying functions are *write*() and *lseek*().

8.2.3.5 ***fgetc*(), *fgets*(), *fread*(), *getc*(), *getchar*(), *gets*(), *scanf*(), *fscanf*()**

These functions may mark the *st_atime* field for update. The *st_atime* field shall be marked for update by the first successful execution of one of these functions that returns data not supplied by a prior call to *ungetc*().

The underlying functions are *read*() and *lseek*().

8.2.3.6 ***fputc*(), *fputs*(), *fwrite*(), *putc*(), *putchar*(), *puts*(), *printf*(), *fprintf*()**

The *st_crime* and *st_mtime* fields of the file shall be marked for update between the successful execution of one of these functions and the next successful completion of a call to either *fflush*() or *fclose*() on the same stream or a call to *exit*() or *abort*().

The underlying functions are *write*() and *lseek*().

If *fwrite*() writes greater than zero bytes, but fewer than requested, the error indicator for the stream shall be set. If the underlying *write*() reports an error, *errno* shall not be modified by *fwrite*(), and the error indicator for the stream shall be set.

If the implementation provides the *vprintf*() and *vfprintf*() functions from the C Standard {2}, they also shall meet the constraints specified in this part of ISO/IEC 9945 for (respectively) *printf*() and *fprintf*().

8.2.3.7 ***fseek*(), *rewind*()**

These functions shall mark the *st_ctime* and *st_mtime* fields of the file for update if the stream was writable and if buffered data had not yet been written to the file.

The underlying functions are *lseek*() and *write*().

If the most recent operation, other than *ftell*(), on a given stream is *fflush*(), the file offset in the underlying open file description shall be adjusted to reflect the location specified by the *fseek*().

8.2.3.8 ***perror*()**

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*.

8.2.3.9 ***tmpfile*()**

The *tmpfile*() function shall allocate a file descriptor as *fopen*() does.

8.2.3.10 ***ftell*()**

The underlying function is *lseek*(). The result of *ftell*() after an *fflush*() shall be the same as the result before the *fflush*(). If the stream is opened in append mode or if the O_APPEND flag is set as a consequence of dealing with other handles on the file, the result of *ftell*() on that stream is unspecified.

8.2.3.11 Error Reporting

If any of the functions above return an error indication, the value of *errno* shall be set to indicate the error condition. If that error condition is one that this part of ISO/IEC 9945 specifies to be detected by one of the corresponding underlying functions, the value of *errno* shall be the same as the value specified for the underlying function.

8.2.3.12 *exit*(), *abort*()

The *exit*() function shall have the effect of *fclose*() on every open stream, with the properties of *fclose*() as described above. The *abort*() function shall also have these effects if the call to *abort*() causes process termination, but shall have no effect on streams otherwise. The C Standard {2} specifies the conditions where *abort*() does or does not cause process termination. For the purposes of that specification, a signal that is blocked shall not be considered caught.

8.2.4 Operations on Files — the *remove*() Function

The *remove*() function shall have the same effect on file times as *unlink*().

8.2.5 Temporary File Name — the *tmpnam*() Function

If the application uses any of the interfaces guaranteed to be available if either {_POSIX_THREAD_SAFE_FUNCTIONS} or {_POSIX_THREADS} is defined, the *tmpnam*() function shall be called with a non-**NULL** parameter.

8.2.6 Stdio Locking Functions

Functions: *flockfile*(), *ftrylockfile*(), *funlockfile*()

8.2.6.1 Synopsis

```
#include <stdio.h>
void flockfile(FILE *file);
int ftrylockfile(FILE *file);
void funlockfile(FILE *file);
```

8.2.6.2 Description

If {_POSIX_THREAD_SAFE FUNCTIONS} is defined:

> The *flockfile*(), *ftrylockfile*(), and *funlockfile*() functions 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 to be executed as a unit.
> The *flockfile*() function is used by a thread to acquire ownership of a (*FILE* *) object.
> The *ftrylockfile*() function is used by a thread to acquire ownership of a (*FILE* *) object if the object is available; *ftrylockfile*() is a nonblocking version of *flockfile*().
> The *funlockfile*() function is used to relinquish the ownership granted to the thread. The behavior is undefined if a thread other than the current owner calls the *funlockfile*() function.
> The implementation shall act 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 is incremented. Otherwise, the calling thread is suspended, waiting for the count to return to zero. Each call to *funlockfile*() decrements the count. This allows matching calls to *flockfile*() [or successful calls to *ftrylockfile*()] and *funlockfile*() to be nested.

All POSIX.1 and C Standard {2} functions that reference (*FILE* *) objects shall behave as if they use *flockfile*() and *funlockfile*() internally to obtain ownership of these (*FILE* *) objects.

Otherwise:

Either the implementation shall support the *flockfile*(), *ftrylockfile*(), and *funlockfile*() functions as described above or the *flockfile*(), *ftrylockfile*(), and *funlockfile*() functions shall not be provided.

8.2.6.3 Returns

None for *flockfile*() and *funlockfile*(). The function *ftrylock*() returns zero for success and nonzero to indicate that the lock cannot be acquired.

8.2.6.4 Errors

None.

8.2.6.5 Cross-References

getc_unlocked(), 8.2.7; *getchar_unlocked*(), 8.2.7; *putc_unlocked*(), 8.2.7; *putchar_unlocked*(), 8.2.7.

8.2.7 Stdio With Explicit Client Locking

Functions: *getc_unlocked*(), *getchar_unlocked*(), *putc_unlocked*(), *putchar_unlocked*()

8.2.7.1 Synopsis

```
#include <stdio.h>
int getc_unlocked(FILE *stream);
int getchar_unlocked(void);
int putc_unlocked(int c, FILE *stream);
int putchar_unlocked(int c);
```

8.2.7.2 Description

If {_POSIX_THREAD_SAFE_FUNCTIONS} is defined:

Versions of the functions *getc*(), *getchar*(), *putc*(), and *putchar*() respectively named *getc_unlocked*(), *getchar_unlocked*(), *putc_unlocked*(), and *putchar_unlocked*() shall be provided, which are functionally identical to the original versions with the exception that they are not required to be implemented in a thread-safe manner. They may only safely be used within a scope protected by *flockfile*() [or *ftrylockfile*()] and *funlockfile*(). These functions may safely be used in a multithreaded program if and only if they are called while the invoking thread owns the (*FILE* *) object, as is the case after a successful call of the *flockfile*() or *ftrylockfile*() functions.

Otherwise:

Either the implementation shall support the *getc_unlocked*(), *getchar_unlocked*(), *putc_unlocked*(), and *putchar_unlocked*() functions as described above or the *getc_unlocked*(), *getchar_unlocked*(), *putc_unlocked*(), and *putchar_unlocked*() functions shall not be provided.

8.2.7.3 Returns

See the C Standard {2}.

8.2.7.4 Errors

This standard does not specify any error conditions that are required to be detected by these interfaces. Some errors may be detected under implementation-defined conditions or as defined by C Standard {2}.

8.2.7.5 Cross-References

getc(), 8.1; *getchar*(), 8.1; *putc*(), 8.1; *putchar*(), 8.1.

8.3 Other C Language Functions

8.3.1 Nonlocal Jumps

Functions: *setjmp*(), *longjmp*(), *sigsetjmp*(), *siglongjmp*()

8.3.1.1 Synopsis

```
#include <setjmp.h>
int sigsetjmp(sigjmp_buf env, int savemask);
void siglongjmp(sigjmp_buf env, int val);
```

8.3.1.2 Description

The *sigsetjmp*() macro shall comply with the definition of the *setjmp*() macro in the C Standard {2}. If the value of the *savemask* argument is not zero, the *sigsetjmp*() function shall also save the current signal mask of the thread (see 3.3.1.1) as part of the calling environment.

The *siglongjmp*() function shall comply with the definition of the *longjmp*() function in the C Standard {2}. If and only if the *env* argument was initialized by a call to the *sigsetjmp*() function with a nonzero *savemask* argument, the *siglongjmp*() function shall restore the saved signal mask.

The effect of a call to *longjmp*() where the initialization of the *jmp_buf* argument was not performed in the calling thread is undefined. The effect of a call to *siglongjmp*() where the initialization of the *sigjmp_buf* argument was not per formed in the calling thread is undefined.

8.3.1.3 Cross-References

sigaction(), 3.3.4; `<signal.h>`, 3.3.1.1; *sigprocmask*(), 3.3.5; *sigsuspend*(), 3.3.7.

8.3.2 Set Time Zone

Function: *tzset*()

8.3.2.1 Synopsis

```
#include <time.h>
void tzset(void);
```

8.3.2.2 Description

The *tzset*() function uses the value of the environment variable **TZ** to set time conversion information used by *localtime*(), *ctime*(), *strftime*(), and *mktime*(). If **TZ** is absent from the environment, implementation-defined default time-zone information shall be used.

The *tzset*() function shall set the external variable *tzname*:

```
    extern char *tzname[2] = { "std" , "dst" } ;
```

where *std* and *dst* are as described in 8.1.1.

8.3.3 Find String Token

Functions: *strtok_r*()

8.3.3.1 Synopsis

```
#include <string.h>
char *strtok_r(char *s, const char *sep, char **lasts);
```

8.3.3.2 Description

If {_POSIX_THREAD_SAFE_FUNCTIONS} is defined:

> The function *strtok_r*() 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* shall point to a null-terminated string, *sep* shall point to a null-terminated string of separator characters, and the value pointed to by *lasts* is ignored. The function *strtok_r*() returns a pointer to the first character of the first token, writes a null character into *s* immediately following the returned token, and updates the pointer to which *lasts* points.
> In subsequent calls, *s* shall be a **NULL** pointer and *lasts* shall be unchanged from the previous call so that subsequent calls will 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 is returned.

Otherwise:

> Either the implementation shall support the *strtok_r*() function as described above or the *strtok_r*() function shall not be provided.

8.3.3.3 Returns

The function *strtok_r*() returns a pointer to the token found, or to a **NULL** pointer when no token is found.

8.3.3.4 Errors

This standard does not specify any error conditions that are required to be detected for the *strtok_r*() function. Some errors may be detected under implementation-defined conditions.

8.3.4 ASCII Time Representation

Functions: *asctime_r*()

8.3.4.1 Synopsis

```
#include <time.h>
char *asctime_r(const struct tm *tm, char *buf);
```

8.3.4.2 Description

If {_POSIX_THREAD_SAFE_FUNCTIONS} is defined:

> The *asctime_r*() function converts the broken-down time in the structure pointed to by *tm* into a string that is placed in the user-supplied buffer pointed to by *buf* (which shall contain at least 26 B) and then returns *buf*.

Otherwise:

> Either the implementation shall support the *asctime_r*() function as described above or the *asctime_r*() function shall not be provided.

8.3.4.3 Returns

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**.

8.3.4.4 Errors

This standard does not specify any error conditions that are required to be detected by the *asctime_r*() function. Some errors may be detected under implementation-defined conditions, or as defined by the C Standard {2}.

8.3.4.5 Cross-References

asctime(), 8.1.1.

8.3.5 Current Time Representation

Functions: *ctime_r*()

8.3.5.1 Synopsis

```
#include <time.h>
char *ctime_r(const time_t *clock, char *buf);
```

8.3.5.2 Description

If {_POSIX_THREAD SAFE FUNCTIONS} is defined:

> The *ctime_r*() function converts the calendar time pointed to by *clock* to local time in exactly the same form as *ctime*(), puts the string into the array pointed to by *buf* (which contains at least 26 B), and returns *buf*. Unlike *ctime*(), the thread-safe version *ctime_r*() is not required to set *tzname*.

Otherwise:

> Either the implementation shall support the *ctime_r*() function as described above or the *ctime_r*() function shall not be provided.

8.3.5.3 Returns

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.

8.3.5.4 Errors

This standard does not specify any error conditions that are required to be detected by the *ctime_r*() function. Some errors may be detected under 617 implementation-defined conditions, or as defined by the C Standard {2}.

8.3.5.5 Cross-References

ctime(), 8.1.

8.3.6 Coordinated Universal Time

Function: *gmtime_r*()

8.3.6.1 Synopsis

```
#include <time.h>
struct tm *gmtime_r(const time_t *clock, struct tm *result);
```

8.3.6.2 Description

If {_POSIX_THREAD_SAFE FUNCTIONS} is defined:

> The *gmtime_r*() function converts the calendar time pointed to by *clock* into a broken-down time expressed as Coordinated Universal Time (UTC). The broken-down time is stored in the structure referred to by *result*. The *gmtime_r*() function also returns the address of the same structure.

Otherwise:

> Either the implementation shall support the *gmtime_r*() function as described above or the *gmtime_r*() function shall not be provided.

8.3.6.3 Returns

Upon successful completion, *gmtime_r*() shall return the address of the structure pointed to by the argument *result*. If an error is detected, or UTC is not available, 637 *gmtime_r*() shall return a **NULL** pointer.

8.3.6.4 Errors

This standard does not specify any additional error conditions that are required to be detected by the *gmtime_r*() function. Some errors may be detected under implementation-defined conditions.

8.3.6.5 Cross-References

gmtime(), 8.1.

8.3.7 Local Time

Functions: *localtime_r*()

8.3.7.1 Synopsis

```
#include <time.h>
struct tm *localtime_r(const time_t *clock, struct tm *result);
```

8.3.7.2 Description

If {_POSIX_THREAD_SAFE_FUNCTIONS} is defined:

> The *localtime_r*() function converts the calendar time pointed to by *clock* into a broken-down time stored in the structure to which *result* points. The *localtime_r*() function also returns a pointer to that same structure. Unlike *localtime*(), the reentrant version is not required to set *tzname*.

Otherwise:

> Either the implementation shall support the *localtime_r*() function as described above or the *localtime_r*() function shall not be provided.

8.3.7.3 Returns

Upon successful completion, *localtime_r*() returns a pointer to the structure pointed to by the argument *result*.

8.3.7.4 Errors

This standard does not specify any error conditions that are required to be detected for the *localtime_r*() function. Some errors may be detected under implementation-defined conditions.

8.3.7.5 Cross-References

localtime(), 8.1.

8.3.8 Pseudo-Random Sequence Generation Functions

Functions: *rand_r*()

8.3.8.1 Synopsis

```
#include <stdlib.h>
int rand_r(unsigned int *seed);
```

8.3.8.2 Description

If {_POSIX_THREAD_SAFE_FUNCTIONS} is defined:

> The *rand_r*() function computes a sequence of pseudo-random integers in the range 0 to RAND_MAX. (The value of the RAND_MAX macro shall be at least 32767.)
> 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.

8.3.8.3 Returns

The *rand_r*() function returns a pseudo-random integer. See the C Standard {2}.

8.3.8.4 Errors

This standard does not specify any error conditions that are required to be detected for the *rand_r*() function. See the C Standard {2}.

8.3.8.5 Cross-References

rand(), 8.1; *srand*(), 8.1.

9. System Databases

9.1 System Databases

The routines described in this section allow an application to access the two system databases that are described below.

The *group* database contains the following information for each group:

1) Group name
2) Numerical group ID
3) List of all users allowed in the group

The *user* database contains the following information for each user:

1) User name
2) Numerical user ID
3) Numerical group ID
4) Initial working directory
5) Initial user program

If the initial user program field is null, the system default is used.

If the initial working directory field is null, the interpretation of that field is implementation defined.

These databases may contain other fields that are unspecified by this part of ISO/IEC 9945.

9.2 Database Access

9.2.1 Group Database Access

Functions: *getgrgid*(), *getgrgid_r*(), *getgrnam*(), *getgrnam_r*()

9.2.1.1 Synopsis

```
#include <sys/types.h>
#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);
struct group *getgrnam(const char *name);
int getgrnam_r(const char *name, struct group *grp, char *buffer,
            size_t bufsize, struct group **result);
```

9.2.1.2 Description

The *getgrgid*() and *getgrnam*() routines both return pointers to an object of type *struct group* containing an entry from the group database with a matching *gid* or *name*. This structure, which is defined in `<grp.h>`, includes the members shown in Table 9.1.

Table 9.1—*group* Structure

Member Type	Member Name	Description
*char **	*gr_name*	The name of the group.
gid_t	*gr_gid*	The numerical group ID.
*char ***	*gr_mem*	A null-terminated vector of pointers to the individual member names.

If {_POSIX_THREAD_SAFE_FUNCTIONS} is defined:

> The *getgrgid_r*() and *getgrnam_r*() functions both update the *group* structure pointed to by *grp* and store a pointer to that structure at the location pointed to by *result*. The structure shall contain an entry from the group database with a matching *gid* or *name*. Storage referenced by the group structure shall be allocated from the memory provided with the *buffer* parameter, which is *bufsize* characters in size. The maximum size needed for this buffer can be determined with the {_SC_GETGR_R_SIZE_MAX} *sysconf*() parameter. A **NULL** pointer is returned at the location pointed to by *result* on error or if the requested entry is not found.

Otherwise:

> Either the implementation shall support the *getgrgid_r*() and *getgrnam_r*() functions as described above or the *getgrgid_r*() and *getgrnam_r*() functions shall not be provided.

9.2.1.3 Returns

A **NULL** pointer is returned from *getgrgid*() and *getgrnam*() on error or if the requested entry is not found.

The return values from *getgrgid*() and *getgrnam*() may point to static data that is overwritten by each call.

If successful, the *getgrgid_r*() and *getgrnam_r*() functions shall return zero. Otherwise, an error number shall be returned to indicate the error.

9.2.1.4 Errors

This part of ISO/IEC 9945 does not specify any error conditions that are required to be detected for the *getgrgid*() or *getgrnam*() functions. Some errors may be detected under conditions that are unspecified by this part of ISO/IEC 9945.

For each of the following conditions, if the condition is detected, the *getgrgid_r*() and *getgrnam_r*() functions shall return the corresponding error number:

[ERANGE]	Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to be referenced by the resulting *group* structure.

9.2.1.5 Cross-References

getlogin(), 4.2.4.

9.2.2 User Database Access

Functions: *getpwuid*(), *getpwuid_r*(), *getpwnam*(), *getpwnam_r*()

9.2.2.1 Synopsis

```
#include <sys/types.h>
#include <pwd.h>
struct passwd *getpwuid (uid_t uid);
int getpwuid_r(uid_t uid, struct passwd *pwd, char *buffer,
            size_t bufsize, struct passwd **result);
struct passwd *getpwnam(const char *name);
int getpwnam_r(const char *name, struct passwd *pwd, char *buffer,
            size_t bufsize, struct passwd **result);
```

9.2.2.2 Description

The *getpwuid*() and *getpwnam*() functions both return a pointer to an object of type *struct passwd* containing an entry from the user database with a matching *uid* or *name*. This structure, which is defined in `<pwd.h>`, includes the members shown in Table 9.2.

Table 9.2—*passwd* Structure

Member Type	Member Name	Description
*char **	*pw_name*	User name.
uid_t	*pw_uid*	User ID number.
gid_t	*pw_gid*	Group ID number.
*char **	*pw_dir*	Initial Working Directory.
*char **	*pw_shell*	Initial User Program.

If {POSIX_THREAD_SAFE_FUNCTIONS} is defined:

> The *getpwuid_r*() and *getpwnam_r*() functions both update the *passwd* structure pointed to by *pwd* and store a pointer to that structure at the loca tion pointed to by *result*. The structure shall contain an entry from the user database with a matching *uid* or *name*. Storage referenced by the structure shall be allocated from the memory provided with the *buffer* parameter, which is *bufsize* characters in size. The maximum size needed for this buffer can be determined with the {_SC_GETPW_R_SIZE_MAX} *sysconf*() parameter. A **NULL** pointer is returned at the location pointed to by *result* on error or if the requested entry is not found.

Otherwise:

> Either the implementation shall support the *getpwuid_r*() and *getpwnam_r*() function as described above or the *getpwuid_r*() and *getpwnam_r*() functions shall not be provided.

9.2.2.3 Returns

A **NULL** pointer is returned from *getpwuid*() and *getpwnam*() on error or if the requested entry is not found.

The return values from *getpwuid*() and *getpwnam*() may point to static data that is overwritten by each call.

If successful, the *getpwuid_r*() and *getpwnam_r*() functions shall return zero. Otherwise, an error number shall be returned to indicate the error.

9.2.2.4 Errors

This part of ISO/IEC 9945 does not specify any error conditions that are required to be detected for the *getpwuid*() or *getpwnam*() functions. Some errors may be detected under conditions that are unspecified by this part of ISO/IEC 9945.

For each of the following conditions, if the condition is detected, the *getpwuid_r*() and *getpwnam_r*() functions shall return the corresponding error number:

[ERANGE]	Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to be referenced by the resulting *passwd* structure.

9.2.2.5 Cross-References

getlogin(), 4.2.4.

10. Data Interchange Format

10.1 Archive/Interchange File Format

A conforming system shall provide a mechanism to copy files from a medium to the file hierarchy and copy files from the file hierarchy to a medium using the interchange formats described here. This part of ISO/IEC 9945 does not define this mechanism.

When this mechanism is used to copy files from the medium by a process without appropriate privileges, the protection information (ownership and access permissions) shall be set in the same fashion that *creat*() would when given the *mode* argument matching the file permissions supplied by the *mode* field of the extended `tar` format or the *c_mode* field of the extended `cpio` format. A process with appropriate privileges shall restore the ownership and the permissions exactly as recorded on the medium, except that the symbolic user and group IDs are used for the `tar` format, as described in 10.1.1.

The *format-creating utility* is used to translate from the file system to the formats defined in this clause. The *format-reading utility* is used to translate from the formats defined in this clause to a file system. The interface to these utilities, including their name or names, is implementation defined.

The headers of these formats are defined to use characters represented in ISO/IEC 646 {1}; however, no restrictions are placed on the contents of the files themselves. The data in a file may be binary data or text represented in any format available to the user. When these formats are used to transfer text at the source level, all characters shall be represented in ISO/IEC 646 {1} International Reference Version (IRV).

The media format and the frames on the media in which the data appear are unspecified by this part of ISO/IEC 9945.

NOTE — Guidelines are given in B.

10.1.1 Extended `tar` Format

An extended `tar` archive tape or file contains a series of blocks. Each block is a fixed-size block of 512 bytes (see below). Although this format may be thought of as being stored on 9-track industry-standard 12,7 mm (0,5 in) magnetic tape, other types of transportable media are not excluded. Each file archived is represented by a header block that describes the file, followed by zero or more blocks that give the contents of the file. At the end of the archive file are two blocks filled with binary zeroes, interpreted as an end-of-archive indicator.

The blocks may be grouped for physical I/O operations. Each group of *n* blocks (where *n* is set by the application utility creating the archive file) may be written with a single *write*() operation. On magnetic tape, the result of this write is a single tape record. The last group of blocks is always at the full size, so blocks after the two zero blocks contain undefined data.

The header block is structured as shown in Table 10.1. All lengths and offsets are in decimal.

Table 10.1—`tar` Header Block

Field Name	Byte Offset	Length (in bytes)
name	0	100
mode	100	8
uid	108	8
gid	116	8
size	124	12
mtime	136	12
chksum	148	8
typeflag	156	1
linkname	157	100
magic	257	6
version	263	2
uname	265	32
gname	297	32
devmajor	329	8
devminor	337	8
prefix	345	155

Symbolic constants used in the header block are defined in the header `<tar.h>` as follows:

```
#define TMAGIC   "ustar" /* ustar and a null */
#define TMAGLEN  6
#define TVERSION "00"    /* 00 and no null */
#define TVERSLEN 2

/* Values used in typeflag field */
#define REGTYPE  '0'     /* Regular file  */
#define AREGTYPE '\0'    /* Regular file  */
#define LNKTYPE  '1'     /* Link          */
#define SYMTYPE  '2'     /* Reserved      */
#define CHRTYPE  '3'     /* Character special */
#define BLKTYPE  '4'     /* Block special */
```

```
#define DIRTYPE  '5'     /* Directory     */
#define FIFOTYPE '6'     /* FIFO special */
#define CONTTYPE '7'     /* Reserved      */

/* Bits used in the mode field - values in octal  */
#define TSUID    04 000  /* Set UID on execution */
#define TSGID    02 000  /* Set GID on execution */
#define TSVTX    01 000  /* Reserved */
                         /* File permissions */
#define TUREAD   00 400  /* Read by owner */
#define TUWRITE  00 200  /* Write by owner */
#define TUEXEC   00 100  /* Execute/Search by owner */
#define TGREAD   00 040  /* Read by group */
#define TGWRITE  00 020  /* Write by group */
#define TGEXEC   00 010  /* Execute/Search by group */
#define TOREAD   00 004  /* Read by other */
#define TOWRITE  00 002  /* Write by other */
#define TOEXEC   00 001  /* Execute/Search by other */
```

All characters are represented in the coded character set of ISO/IEC 646 {1}. For maximum portability between implementations, names should be selected from characters represented by the portable filename character set as 8-bit characters with most significant bit zero. If an implementation supports the use of characters outside the portable filename character set in names for files, users, and groups, one or more implementation-defined encodings of these characters shall be provided for interchange purposes. However, the format-reading utility shall never create file names on the local system that cannot be accessed via the functions described previously in this part of ISO/IEC 9945; see 5.3.1, 5.6.2, 5.2.1, 6.5.2, and 5.1.2. If a file name is found on the medium that would create an invalid file name, the implementation shall define if the data from the file is stored on the file hierarchy and under what name it is stored. A format-reading utility may choose to ignore these files as long as it produces an error indicating 104 that the file is being ignored.

Each field within the header block is contiguous; that is, there is no padding used. Each character on the archive medium is stored contiguously.

The fields *magic*, *uname*, and *gname* are null-terminated character strings. The fields *name*, *linkname*, and *prefix* are null-terminated character strings except when all characters in the array contain nonnull characters including the last character. The *version* field is two bytes containing the characters `"00"` (zero-zero). The *typeflag* contains a single character. All other fields are leading zero-filled octal numbers using digits from ISO/IEC {1} IRV. Each numeric field is terminated by one or more space or null characters.

The *name* and the *prefix* fields produce the pathname of the file. The hierarchical relationship of the file is retained by specifying the pathname as a path prefix, and a slash character and filename as the suffix. A new pathname is formed, if *prefix* is not an empty string (its first character is not null), by concatenating *prefix* (up to the first null character), a slash character, and *name*; otherwise, *name* is used alone. In either case, *name* is terminated at the first null character. If *prefix* is an empty string, it is simply ignored. In this manner, pathnames of at most 256 characters can be supported. If a pathname does not fit in the space provided, the format-creating utility shall notify the user of the error, and no attempt shall be made by the format-creating utility to store any part of the file—header or data—on the medium.

The *linkname* field, described below, does not use the *prefix* to produce a pathname. As such, a *linkname* is limited to 100 characters. If the name does not fit in the space provided, the format-creating utility shall notify the user of the error, and the utility shall not attempt to store the link on the medium.

The *mode* field provides 9 bits specifying file permissions and 3 bits to specify the set UID, set GID, and TSVTX modes. Values for these bits were defined previously. When appropriate privilege is required to set one of these mode bits, and the user restoring the files from the archive does not have the appropriate privilege, the mode bits for which the user does not have appropriate privilege shall be ignored. Some of the mode bits in the archive format are not mentioned elsewhere in this part of ISO/IEC 9945 . If the implementation does not support those bits, they may 136 be ignored.

The *uid* and *gid* fields are the user and group ID of the owner and group of the file, respectively.

The *size* field is the size of the file in bytes. If the *typeflag* field is set to specify a file to be of type LNKTYPE or SYMTYPE, the *size* field shall be specified as zero. If the *typeflag* field is set to specify a file of type DIRTYPE, the *size* field is interpreted as described under the definition of that record type. No data blocks are 143 stored for LNKTYPE, SYMTYPE, or DIRTYPE. If the *typeflag* field is set to CHRTYPE, BLKTYPE, or FIFOTYPE, the meaning of the *size* field is unspecified by this part of ISO/IEC 9945, and no data blocks are stored on the medium. Additionally, for FIFOTYPE, the *size* field shall be ignored when reading. If the *typeflag* field is set to any other value, the number of blocks written following the header is (*size*+511)/512, ignoring any fraction in the result of the division.

The *mtime* field is the modification time of the file at the time it was archived. It is the ISO/IEC 646 {1} representation of the octal value of the modification time 151 obtained from the *stat*() function.

The *chksum* field is the ISO/IEC 646 {1} IRV representation of the octal value of the simple sum of all bytes in the header block. Each 8-bit byte in the header is treated as an unsigned value. These values are added to an unsigned integer, initialized to zero, the precision of which shall be no less than 17 bits. When calculating the checksum, the *chksum* field is treated as if it were all blanks.

The *typeflag* field specifies the type of file archived. If a particular implementation does not recognize the type, or the user does not have appropriate privilege to create that type, the file shall be extracted as if it were a regular file if the file type is defined to have a meaning for the size field that could cause data blocks to be written on the medium (see the previous description for *size*). If conversion to an ordinary file occurs, the format-reading utility shall produce an error indicating that the conversion took place. All of the *typeflag* fields are coded in ISO/IEC 646 {1} IRV:

`'0'` Represents a regular file. For backward compatibility, a *typeflag* value of binary zero (`'\0'`) should be recognized as meaning a regular file when extracting files from the archive. Archives written with this version of the archive file format shall create regular files with a *typeflag* value of ISO/IEC 646 {1} IRV `'0'`.

`'1'` Represents a file linked to another file, of any type, previously archived. Such files are identified by each file having the same device and file serial number. The linked-to name is specified in the *linkname* field with a null terminator if it is less than 100 bytes in length.

`'2'` Reserved to represent a link to another file, of any type, whose device or file serial number differs. This is provided for systems that support linked files whose device or file serial numbers differ, and should be treated as a type `'1'` file if this extension does not exist.

`'3'`,`'4'` Represent character special files and block special files respectively. In this case the *devmajor* and *devminor* fields shall contain information defining the device, the format of which is unspecified by this part of ISO/IEC 9945. Implementations may map the device specifications to their own local specification or may ignore the entry.

`'5'` Specifies a directory or subdirectory. On systems where disk allocation is performed on a directory basis, the *size* field shall contain the maximum number of bytes (which may be rounded to the nearest disk block allocation unit) that the directory may hold. A *size* field of zero indicates no such limiting. Systems that do not support limiting in this manner should ignore the *size* field.

`'6'` Specifies a FIFO special file. Note that the archiving of a FIFO file archives the existence of this file and not its contents.

`'7'` Reserved to represent a file to which an implementation has associated some high performance attribute. Implementations without such extensions should treat this file as a regular file (type `'0'`).

`'A'–'Z'` The letters A through Z are reserved for custom implementations. All other values are reserved for specification in future revisions of this part of ISO/IEC 9945.

The *magic* field is the specification that this archive was output in this archive format. If this field contains TMAGIC, the *uname* and *gname* fields shall contain the ISO/IEC 646 {1} IRV representation of the owner and group of the file respectively (truncated to fit, if necessary). When the file is restored by a privileged, protection-preserving version of the utility, the password and group files shall be scanned for these names. If found, the user and group IDs contained within these files shall be used rather than the values contained within the *uid* and *gid* fields.

The encoding of the header is designed to be portable across machines.

10.1.1.1 Cross-References

`<grp.h>`, 9.2.1.1; `<pwd.h>`, 9.2.2.1; `<sys/stat.h>`, 5.6.1; *stat*(), 5.6.2; `<unistd.h>`, 2.9.

10.1.2 Extended `cpio` Format

The byte-oriented `cpio` archive format is a series of entries, each comprised of a header that describes the file, the name of the file, and then the contents of the file.

An archive may be recorded as a series of fixed-size blocks of bytes. This blocking shall be used only to make physical I/O more efficient. The last group of blocks is always at the full size.

For the byte-oriented `cpio` archive format, the individual entry information must be in the order indicated and described by Table 10.2.

Table 10.2—Byte-Oriented `cpio` Archive Entry

Header		
Field Name	**Length (in bytes)**	**Interpreted as**
c_magic	6	Octal number
c_dev	6	Octal number
c_ino	6	Octal number
c_mode	6	Octal number
c_uid	6	Octal number
c_gid	6	Octal number
c_nlink	6	Octal number
c_rdev	6	Octal number
c_mtime	11	Octal number
c_namesize	6	Octal number
c_filesize	11	Octal number
File Name		
Field Name	**Length**	**Interpreted as**
c_name	*c_namesize*	Pathname string
File Data		
Field Name	**Length**	**Interpreted as**
c_filedata	*c_filesize*	Data

10.1.2.1 `cpio` Header

For each file in the archive, a header as defined previously shall be written. The information in the header fields shall be written as streams of ISO/IEC 646 {1} characters interpreted as octal numbers. The octal numbers are extended to the necessary length by appending ISO/IEC 646 {1} IRV zeros at the most-significant-digit end of the number; the result is written to the stream of bytes most-significant-digit first. The fields shall be interpreted as follows:

1) *c_magic* shall identify the archive as being a transportable archive by containing the magic bytes as defined by MAGIC (`070707`).
2) *c_dev* and *c_ino* shall contain values that uniquely identify the file within the archive (i.e., no files shall contain the same pair of *c_dev* and *c_ino* values unless they are links to the same file). The values shall be determined in an unspecified manner.
3) *c_mode* shall contain the file type and access permissions as defined in Table 10.3.
4) *c_uid* shall contain the user ID of the owner.
5) *c_gid* shall contain the group ID of the group.

Table 10.3—Values for `cpio` *c_mode* Field

File Permissions		
Name	**Value**	**Indicates**
C_IRUSR	000 400	Read by owner.
C_IWUSR	000 200	Write by owner.
C_IXUSR	000 100	Execute by owner.
C_IRGRP	000 040	Read by group.
C_IWGRP	000 020	Write by group.
C_IXGRP	000 010	Execute by group.
C_IROTH	000 004	Read by others.
C_IWOTH	000 002	Write by others.
C_IXOTH	000 001	Execute by others.
C_ISUID	004 000	Set *uid.*
C_ISGID	002 000	Set *gid.*
C_ISVTX	001 000	Reserved.
File Type		
Name	**Value**	**Indicates**
C_ISDIR	040 000	Directory.
C_ISFIFO	010 000	FIFO.
C_ISREG	0100 000	Regular file.
C_ISBLK	060 000	Block special file.
C_ISCHR	020 000	Character special file.
C_ISCTG	0110 000	Reserved.
C_ISLNK	0120 000	Reserved.
C_ISSOCK	0140 000	Reserved.

6) *c_nlink* shall contain the number of links referencing the file at the time the archive was created.
7) *c_rdev* shall contain implementation-defined information for character or block special files.
8) *c_mtime* shall contain the latest time of modification of the file at the time the archive was created.
9) *c_namesize* shall contain the length of the pathname, including the terminating null byte.
10) *c_filesize* shall contain the length of the file in bytes. This is the length of the data section following the header structure.

10.1.2.2 `cpio` File Name

c_name shall contain the pathname of the file. The length of this field in bytes is the value of *c_namesize*. If a file name is found on the medium that would create an invalid pathname, the implementation shall define if the data from the file is stored on the file hierarchy and under what name it is stored.

All characters are represented in ISO/IEC 646 {1} IRV. For maximum portability between implementations, names should be selected from characters represented by the portable filename character set as 8-bit characters most significant bit zero. If an implementation supports the use of characters outside the portable filename character set in names for files, users, and groups, one or more implementation-defined encodings of these characters shall be provided for interchange purposes. However, the format-reading utility shall never create file names on the local system that cannot be accessed via the functions described previously in this part of ISO/IEC 9945; see *open*(), *stat*(), *chdir*(), *fcntl*(), and *opendir*(). If a file name is found on the medium that would create an invalid file name, the implementation shall define if the data from the file is stored on the local file system and under what name it is stored. A format-reading utility may choose to ignore these files as long as it produces an error indicating that the file is being ignored.

10.1.2.3 `cpio` File Data

Following *c_name*, there shall be *c_filesize* bytes of data. Interpretation of such data shall occur in a manner dependent on the file. If *c_filesize* is zero, no data shall be contained in *c_filedata*.

10.1.2.4 `cpio` Special Entries

FIFO special files, directories, and the trailer are recorded with *c_filesize* equal to zero. For other special files, *c_filesize* is unspecified by this part of ISO/IEC 9945. The header for the next file entry in the archive shall be written directly after the last byte of the file entry preceding it. A header denoting the file name "`TRAILER!!!`" shall indicate the end of the archive; the contents of bytes in the last block of the archive following such a header are undefined.

10.1.2.5 `cpio` Values

Values needed by the `cpio` archive format are described in Table 10.3.

C_ISDIR, C_ISFIFO, and C_ISREG shall be supported on a system conforming to this part of ISO/IEC 9945; additional values defined previously are reserved for compatibility with existing systems. Additional file types may be supported; however, such files should not be written on archives intended for transport to portable systems.

C_ISVTX, C_ISCTG, C_ISLNK, and C_ISSOCK have been reserved by this part of ISO/IEC 9945 to retain compatibility with some existing implementations.

When restoring from an archive:

1) If the user does not have the appropriate privilege to create a file of the specified type, the format-interpreting utility shall ignore the entry and issue an error to the standard error output.
2) Only regular files have data to be restored. Presuming a regular file meets any selection criteria that might be imposed on the format-reading utility by the user, such data shall be restored.

3) If a user does not have appropriate privilege to set a particular mode flag, the flag shall be ignored. Some of the mode flags in the archive format are not mentioned elsewhere in this part of ISO/IEC 9945. If the implementation does not support those flags, they may be ignored.

10.1.2.6 Cross-References

`<grp.h>`, 9.2.1.1; `<pwd.h>`, 9.2.2.1; `<sys/stat.h>`, 5.6.1; *chmod*(), 5.6.4; *link*(), 5.3.4; *mkdir*(), 5.4.1; *read*(), 6.4.1; *stat*(), 5.6.2.

10.1.3 Multiple Volumes

It shall be possible for data represented by the Archive/Interchange File Format to reside in more than one file.

The format is considered a stream of bytes. An end-of-file (or equivalently an end-of-media) condition may occur between any two bytes of the logical byte stream. If this condition occurs, the byte following the end-of-file will be the first byte on the next file. The format-reading utility shall, in an implementation defined manner, determine what file to read as the next file.

11. Synchronization

The facilities described in this clause provide synchronization via counting semaphores, mutexes, and condition variables.

11.1 Semaphore Characteristics

The header `<semaphore.h>` shall define the type *sem_t*, used in performing semaphore operations. The type *sem_t* is used to represent semaphores. The semaphore may be implemented using a file descriptor. In that case, applications shall be able to open up to at least a total of {OPEN_MAX} files and semaphores; see 5.3.1.

Inclusion of the `<semaphore.h>` header may make visible the symbols allowed by this part of ISO/IEC 9945 to be in the headers `<sys/types.h>` and `<fcntl.h>`.

11.2 Semaphore Functions

11.2.1 Initialize an Unnamed Semaphore

Function: *sem_init*()

11.2.1.1 Synopsis

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

11.2.1.2 Description

If {_POSIX_SEMAPHORES} is defined:

> The *sem_init*() function is used to initialize the unnamed semaphore referred to by *sem*. The value of the initialized semaphore is *value*. Following a successful call to *sem_init*(), the semaphore may be used in subsequent calls to *sem_wait*(), *sem_trywait*(), *sem_post*(), and *sem_destroy*(). This semaphore remains usable until the semaphore is destroyed.

If the *pshared* argument has a nonzero value, then the semaphore is shared between processes; in this case, any process that can access the semaphore *sem* can use *sem* for performing *sem_wait*(), *sem_trywait*(), *sem_post*(), and *sem_destroy*() operations.
Only *sem* itself may be used for performing synchronization. The result of referring to copies of *sem* in calls to *sem_wait*(), *sem_trywait*(), *sem_post*(), and *sem_destroy*(), is undefined.
If the *pshared* argument is zero, then the semaphore is shared between threads of the process; any thread in this process can use *sem* for performing *sem_wait*(), *sem_trywait*(), *sem_post*(), and *sem_destroy*() operations. The use of the semaphore by threads other than those created in the same process is undefined.
Attempting to initialize an already initialized semaphore results in undefined behavior.

Otherwise:

Either the implementation shall support the *sem_init*() function as described above or the *sem_init*() function shall fail.

11.2.1.3 Returns

Upon successful completion, the function shall initialize the semaphore in *sem*. Otherwise, it shall return –1 and set *errno* to indicate the error.

11.2.1.4 Errors

If any of the following conditions occur, the *sem_init*() function shall return –1 and set *errno* to the corresponding value:

[EINVAL]	The *value* argument exceeds {SEM_VALUE_MAX}.
[ENOSPC]	A resource required to initialize the semaphore has been exhausted.
	The limit on semaphores ({SEM_NSEMS_MAX}) has been reached.
[ENOSYS]	The function *sem_init*() is not supported by this implementation.
[EPERM]	The process lacks the appropriate privileges to initialize the semaphore.

11.2.1.5 Cross-References

`<semaphore.h>`, 11.1; *sem_destroy*(), 11.2.2; *sem_post*(), 11.2.7; *sem_trywait*(), 11.2.6; *sem_wait*(), 11.2.6.

11.2.2 Destroy an Unnamed Semaphore

Function: *sem_destroy*()

11.2.2.1 Synopsis

```
#include <semaphore.h>
int sem_destroy(sem_t *sem);
```

11.2.2.2 Description

If {_POSIX_SEMAPHORES} is defined:

The *sem_destroy*() function is used to destroy the unnamed semaphore indicated by *sem*. Only a semaphore that was created using *sem_init*() may be destroyed using *sem_destroy*(); the effect of calling *sem_destroy*() with a named semaphore is undefined. The effect of subsequent use of the semaphore *sem* is undefined until *sem* is re-initialized by another call to *sem_init*().

It shall be safe to destroy an initialized semaphore upon which no threads are currently blocked. The effect of destroying a semaphore upon which other threads are currently blocked is undefined.

Otherwise:

Either the implementation shall support the *sem_destroy*() function as described above or the *sem_destroy*() function shall fail.

11.2.2.3 Returns

Upon successful completion, a value of zero shall be returned. Otherwise, a value of –1 is returned and *errno* shall be set to indicate the error.

11.2.2.4 Errors

If any of the following conditions occur, the *sem_destroy*() function shall return –1 ανδ σετ *errno* to the corresponding value:

[EINVAL]	The *sem* argument is not a valid semaphore.
[ENOSYS]	The function *sem_destroy*() is not supported by this implementation.

For each of the following conditions, if the condition is detected, the *sem_destroy*() function shall return –1 ανδ σετ *errno* to the corresponding value:

[EBUSY]	There are currently processes blocked on the semaphore.

11.2.2.5 Cross-References

`<semaphore.h>`, 11.1; *sem_init*(), 11.2.1; *sem_open*(), 11.2.3.

11.2.3 Initialize/Open a Named Semaphore

Function: *sem_open*()

11.2.3.1 Synopsis

```
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag, ...);
```

11.2.3.2 Description

If {_POSIX_SEMAPHORES} is defined:

The *sem_open*() function establishes a connection between a named semaphore and a process. Following a call to *sem_open*() with semaphore name *name*, the process may reference the semaphore associated with *name* using the address returned from the call. This semaphore may be used in subsequent calls to *sem_wait*(), *sem_trywait*(), *sem_post*(), and *sem_close*(). The semaphore remains usable by this process until the semaphore is closed by a successful call to *sem_close*(), *_exit*(), or a *exec* function.
The *oflag* argument controls whether the semaphore is created or merely accessed by the call to *sem_open*(). The following flag bits may be set in *oflag*:

O_CREAT	This flag is used to create a semaphore if it does not already exist. If O_CREAT is set and the semaphore already exists, then O_CREAT has no effect, except as noted under O_EXCL. Otherwise, *sem_open*() creates a named semaphore. The O_CREAT flag

requires a third and a fourth argument: *mode*, which is of type *mode_t*, and *value*, which is of type *unsigned int*. The semaphore is created with an initial value of *value*. Valid initial values for semaphores shall be less than or equal to {SEM_VALUE_MAX}.

The user ID of the semaphore shall be set to the effective user ID of the process; the group ID of the semaphore shall be set to a system default group ID or to the effective group ID of the process. The permission bits of the semaphore 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 specified, the effect is unspecified.

After the semaphore named *name* has been created by *sem_open*() with the O_CREAT flag, other processes can connect to the semaphore by calling *sem_open*() with the same value of *name*.

O_EXCL If O_EXCL and O_CREAT are set, *sem_open*() shall fail if the semaphore *name* exists. The check for the existence of the semaphore and the creation of the semaphore if it does not exist shall be atomic with respect to other processes executing *sem_open*() with O_EXCL and O_CREAT set. If O_EXCL is set and O_CREAT is not set, the effect is undefined.

If flags other than O_CREAT and O_EXCL are specified in the *oflag* parameter, the effect is unspecified.

The *name* argument points to a string naming a semaphore object. It is unspecified whether the name appears in the file system and is visible to functions that take pathnames as arguments. The *name* argument shall conform to the construction rules for a pathname. If *name* begins with the slash character, then processes calling *sem_open*() with the same value of name shall refer to the same semaphore object, as long as that name has not been removed. If *name* does not begin with the slash character, the effect is implementation defined. The interpretation of slash characters other than the leading slash character in *name* is implementation defined.

If a process makes multiple successful calls to *sem_open*() with the same value for *name*, the same semaphore address shall be returned for each such successful call, provided that there have been no calls to *sem_unlink*() for this semaphore.

References to copies of the semaphore produce undefined results.

Otherwise:

Either the implementation shall support the *sem_open*() function as described above or the *sem_open*() function shall fail.

11.2.3.3 Returns

Upon successful completion, the 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 shall be defined in the header `<semaphore.h>`. No successful return from *sem_open*() shall return the value SEM_FAILED.

11.2.3.4 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. The implementation shall document under what circumstances this error may be returned.
	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.
[ENAMETOOLONG]	The length of the *name* string exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.
[ENFILE]	Too many semaphores are currently open in the system.
[ENOENT]	O_CREAT is not set and the named semaphore does not exist.
[ENOSPC]	There is insufficient space for the creation of the new named semaphore.
[ENOSYS]	The function *sem_open*() is not supported by this implementation.

11.2.3.5 Cross-References

`<semaphore.h>`, 11.1; *sem_close*(), 11.2.4; *sem_post*(), 11.2.7; *sem_trywait*(), 11.2.6; *sem_unlink*(), 11.2.5; *sem_wait*(), 11.2.6.

11.2.4 Close a Named Semaphore

Function: *sem_close*()

11.2.4.1 Synopsis

```
#include <semaphore.h>
int sem_close(sem_t *sem);
```

11.2.4.2 Description

If {_POSIX_SEMAPHORES} is defined:

> The *sem_close*() function is used to indicate that the calling process is finished using the named semaphore indicated by *sem*. The effects of calling *sem_close*() for an unnamed semaphore [one created by *sem_init*()] are undefined. The *sem_close*() function shall deallocate [that is, make available for reuse by a subsequent *sem_open*() by this process] any system resources allocated by the system for use by this process for this semaphore. The effect of subsequent use of the semaphore indicated by *sem* by this process is undefined. If the semaphore has not been removed with a successful call to *sem_unlink*(), then *sem_close*() shall have no effect on the state of the semaphore. If the *sem_unlink*() function has been successfully invoked for *name* after the most recent call to *sem_open*() with O_CREAT for this semaphore, then when all processes that have opened the semaphore close it, the semaphore shall no longer be accessible.

Otherwise:

> Either the implementation shall support the *sem_close*() function as described above or the *sem_close*() function shall fail.

11.2.4.3 Returns

Upon successful completion, a value of zero shall be returned. Otherwise, a value of –1 is returned and *errno* shall be set to indicate the error.

11.2.4.4 Errors

If any of the following conditions occur, the *sem_close*() function shall return –1 and set *errno* to the corresponding value:

[EINVAL]	The *sem* argument is not a valid semaphore descriptor.
[ENOSYS]	The function *sem_close*() is not supported by this implementation.

11.2.4.5 Cross-References

`<semaphore.h>`, 11.1; *sem_init*(), 11.2.1; *sem_open*(), 11.2.3; *sem_unlink*(), 11.2.5.

11.2.5 Remove a Named Semaphore

Function: *sem_unlink*()

11.2.5.1 Synopsis

```
#include <semaphore.h>
int sem_unlink(const char *name);
```

11.2.5.2 Description

If {_POSIX_SEMAPHORES} is defined:

> 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 shall be postponed until all references to the semaphore have destroyed by calls to *sem_close*(), *_exit*(), or *exec*. Calls to *sem_open*() to re-create or re-connect to the semaphore shall 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.

Otherwise:

> Either the implementation shall support the *sem_unlink*() function as described above or the *sem_unlink*() function shall fail.

11.2.5.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, the semaphore shall not be changed by this function call, and the function shall return a value of –1 and set *errno* to indicate the error.

11.2.5.4 Errors

If any of the following conditions occur, the *sem_unlink*() function shall return –1 and set *errno* to the corresponding value:

[EACCES]	Permission is denied to unlink the named semaphore.
[ENAMETOOLONG]	The length of the *name* string exceeds {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.
[ENOENT]	The named semaphore does not exist.

[ENOSYS]	The function *sem_unlink*() is not supported by this implementation.

11.2.5.5 Cross-References

`<semaphore.h>`, 11.1; *sem_close*(), 11.2.4; *sem_open*(), 11.2.3.

11.2.6 Lock a Semaphore

Functions: *sem_wait*(), *sem_trywait*()

11.2.6.1 Synopsis

```
#include <semaphore.h>
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
```

11.2.6.2 Description

If {_POSIX_SEMAPHORES} is defined:

> The *sem_wait*() function locks the semaphore referenced by *sem* by performing the semaphore lock operation on that semaphore. If the semaphore value is currently zero, then the calling thread shall not return from the call to *sem_wait*() until it either locks the semaphore or the call is interrupted by a signal. The *sem_trywait*() function locks the semaphore referenced by *sem* only if the semaphore is currently not locked; that is, if the semaphore value is currently positive. Otherwise, it does not lock the semaphore.
> Upon successful return, the state of the semaphore is locked and shall remain locked until the *sem_post*() function is executed and returns successfully.
> The *sem_wait*() function shall be interruptible by the delivery of a signal.

Otherwise:

> Either the implementation shall support the *sem_wait*() and *sem_trywait*() functions as described above or each of the *sem_wait*() and *sem_trywait*() functions shall fail.

11.2.6.3 Returns

The *sem_wait*() and *sem_trywait*() functions return zero if the calling process successfully performed the semaphore lock operation on the semaphore designated by *sem*. If the call was unsuccessful, the state of the semaphore shall be unchanged, and the function shall return a value of –1 and set *errno* to indicate the error.

11.2.6.4 Errors

If any of the following conditions occur, the *sem_wait*() and *sem_trywait*() functions shall return –1 and set *errno* to the corresponding value:

[EAGAIN]	The semaphore was already locked, so it cannot be immediately locked by the *sem_trywait*() operation [*sem_trywait*() only].
[EINVAL]	The *sem* argument does not refer to a valid semaphore.
[ENOSYS]	The functions *sem_wait*() and *sem_trywait*() are not supported by this implementation.

For each of the following conditions, if the condition is detected, the *sem_wait*() and *sem_trywait*() functions shall return –1 and set *errno* to the corresponding value:

[EDEADLK]	A deadlock condition was detected.
[EINTR]	A signal interrupted this function. It shall be documented in the system documentation whether this error is returned.

11.2.6.5 Cross-References

`<semaphore.h>`, 11.1; *sem_post*(), 11.2.7.

11.2.7 Unlock a Semaphore

Function: *sem_post*()

11.2.7.1 Synopsis

```
#include <semaphore.h>
int sem_post(sem_t *sem);
```

11.2.7.2 Description

If {_POSIX_SEMAPHORES} is defined:

The *sem_post*() function unlocks the semaphore referenced by *sem* by performing the semaphore unlock operation on that semaphore.
If the semaphore value resulting from this operation is positive, then no threads were blocked waiting for the semaphore to become unlocked; the semaphore value is simply incremented.
If the value of the semaphore resulting from this operation is zero, then one of the threads blocked waiting for the semaphore shall be allowed to return successfully from its call to *sem_wait*(). If the symbol {_POSIX_PRIORITY_SCHEDULING} is defined, the thread to be unblocked shall be chosen in a manner appropriate to the scheduling policies and parameters in effect for the blocked threads. In the case of the schedulers SCHED_FIFO and SCHED_RR, the highest priority waiting thread shall be unblocked, and if there is more than one highest priority thread blocked waiting for the semaphore, then the highest priority thread that has been waiting the longest shall be unblocked. If the symbol {_POSIX_PRIORITY_SCHEDULING} is not defined, the choice of a thread to unblock is unspecified.
The *sem_post*() function shall be reentrant with respect to signals and may be invoked from a signal-catching function.

Otherwise:

Either the implementation shall support the *sem_post*() function as described above or the *sem_post*() function shall fail.

11.2.7.3 Returns

If successful, the *sem_post*() function shall return zero; otherwise the function shall return −1 and set *errno* to indicate the error.

11.2.7.4 Errors

If any of the following conditions occur, the *sem_post*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL]	The *sem* does not refer to a valid semaphore.
[ENOSYS]	The function *sem_post*() is not supported by this implementation.

11.2.7.5 Cross-References

`<semaphore.h>`, 11.1; *sem_trywait*(), 11.2.6; *sem_wait*(), 11.2.6.

11.2.8 Get the Value of a Semaphore

Function: *sem_getvalue*()

11.2.8.1 Synopsis

```
#include <semaphore.h>
int sem_getvalue(sem_t *sem, int *sval);
```

11.2.8.2 Description

If {_POSIX_SEMAPHORES} is defined:

> The *sem_getvalue*() function updates the location referenced by the *sval* argument to have the value of the semaphore referenced by *sem* without affecting the state of the semaphore. The updated value represents an actual semaphore value that occurred at some unspecified time during the call, but it need not be the actual value of the semaphore when it is returned to the calling process.
> If *sem* is locked, then the value returned by *sem_getvalue*() shall be either zero or a negative number whose absolute value represents the number of processes waiting for the semaphore at some unspecified time during the call.

Otherwise:

> Either the implementation shall support the *sem_getvalue*() function as described above or the *sem_getvalue*() function shall fail.

11.2.8.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, the function shall return a value of –1 and set *errno* to indicate the error.

11.2.8.4 Errors

If any of the following conditions occur, the *sem_getvalue*() function shall return –1 and set *errno* to the corresponding value:

[EINVAL]	The *sem* argument does not refer to a valid semaphore.
[ENOSYS]	The function *sem_getvalue*() is not supported by this implementation.

11.2.8.5 Cross-References

`<semaphore.h>`, 11.1; *sem_post*() 11.2.7; *sem_trywait*() 11.2.6; *sem_wait*() 11.2.6.

11.3 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 shall be determined by the scheduling policy. See 13.6 for full details.

A thread becomes the owner of a mutex, *m*, when either

1) It returns successfully from *pthread_mutex_lock*() with *m* as the *mutex* argument, or
2) It returns successfully from *pthread_mutex_trylock*() with *m* as the *mutex* argument, or
3) It returns (successfully or not) from *pthread_cond_wait*() with *m* as the *mutex* argument (except as explicitly indicated otherwise for certain errors), or
4) 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 remains the owner of *m* until it either

1) Executes *pthread_mutex_unlock*() with *m* as the *mutex* argument, or
2) Blocks in a call to *pthread_cond_wait*() with *m* as the *mutex* argument, or
3) Blocks in a call to 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.

11.3.1 Mutex Initialization Attributes

Functions: *pthread_mutexattr_init*(), *pthread_mutexattr_destroy*(), *pthread_mutexattr_getpshared*(), *pthread_mutexattr_setpshared*()

11.3.1.1 Synopsis

```
#include <pthread.h>
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
int pthread_mutexattr_getpshared(const pthread_mutexattr_t *attr,
            int *pshared);
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
            int pshared);
```

11.3.1.2 Description

If {_POSIX_THREADS} is defined:

The function *pthread_mutexattr_init*() initializes a mutex attributes object *attr* with a default value for all of the attributes defined by the implementation.
The effect of initializing an already initialized mutex attributes object is undefined.
After a mutex attributes object has been used to initialize one or more mutexes, any function affecting the attributes object (including destruction) does not affect any previously initialized mutexes.
The *pthread_mutexattr_destroy*() function destroys 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 mutex attributes object can be reinitialized using *pthread_mutexattr_init*(); the results of otherwise referencing the object after it has been destroyed are undefined.
If the symbol {_POSIX_THREAD_PROCESS_SHARED} is defined, the implementation shall provide the attribute process-shared and the associated functions *pthread_mutexattr_getpshared*() and *pthread_mutexattr_setpshared*(). If this symbol is not defined, then the process-shared attribute and these functions are not supported. 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.
The *pthread_mutexattr_setpshared*() function is used to set the process-shared attribute in an initialized attributes object referenced by *attr*. The *pthread_mutexattr_getpshared*() function obtains the value of the process-shared attribute from the attributes object referenced by *attr*.

Otherwise:

Either the implementation shall support the *pthread_mutexattr_init*(), *pthread_mutexattr_destroy*(), *pthread_mutexattr_getpshared*(), and *pthread_mutexattr_setpshared*() functions as described above or the *pthread_mutexattr_init*(), *pthread_mutexattr_destroy*(), *pthread_mutexattr_getpshared*(), and *pthread_mutexattr_setpshared*() functions shall not be provided.

See 13.4.1 for full details of attributes related to scheduling policies.

Additional attributes, their default values, and the names of the associated functions to get and set those attribute values are implementation-defined.

11.3.1.3 Returns

Upon successful completion, *pthread_mutexattr_init*(), *pthread_mutexattr_destroy*(), and *pthread_mutexattr_-setpshared*() shall return zero. Otherwise, an error number shall be returned to indicate the error.

Upon successful completion, the *pthread_mutexattr_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.

11.3.1.4 Errors

For each of the following conditions, if the condition is detected, the *pthread_mutexattr_init*() function shall return the corresponding error number:

[ENOMEM]	Insufficient memory exists to initialize the mutex attributes object.

For each of the following conditions, if the condition is detected, the *pthread_mutexattr_destroy*(), *pthread_mutexattr_getpshared*(), and *pthread_mutexattr_setpshared*() functions shall return the corresponding error number:

[EINVAL]	The value specified by *attr* is invalid.

For each of the following conditions, if the condition is detected, the *pthread_mutexattr_setpshared*() function shall return the corresponding error number:

[EINVAL]	The new value specified for the attribute is outside the range of legal values for that attribute.

11.3.1.5 Cross-References

pthread_create(), 16.2.2; *pthread_mutex_init*(), 11.3.2; *pthread_cond_init*(), 11.4.2.

11.3.2 Initializing and Destroying a Mutex

Functions: *pthread_mutex_init*(), *pthread_mutex_destroy*()

11.3.2.1 Synopsis

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex,
            const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

11.3.2.2 Description

If {_POSIX_THREADS} is defined:

> The *pthread_mutex_init*() function initializes the mutex referenced by *mutex* with attributes specified by *attr*. If *attr* is **NULL**, the default mutex attributes are used; the effect is the same as passing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked.
> Attempting to initialize an already initialized mutex results in undefined behavior.
> The *pthread_mutex_destroy*() function destroys 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.
> 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 the parameter *attr* specified as **NULL**, except that no error checks are performed.

Otherwise:

> Either the implementation shall support the *pthread_mutex_init*() and *pthread_mutex_destroy*() functions as described above or the *pthread_mutex_init*() and *pthread_mutex_destroy*() functions shall not be provided.

11.3.2.3 Returns

If successful, the *pthread_mutex_init*() and *pthread_mutex_destroy*() 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 mutex specified by *mutex.*

11.3.2.4 Errors

If any of the following conditions occur, the *pthread_mutex_init*() function shall return the corresponding error number:

[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.

For each of the following conditions, if the condition is detected, the *pthread_mutex_init*() function shall return the corresponding error number:

[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.

For each of the following conditions, if the condition is detected, the *pthread_mutex_destroy*() function shall return the corresponding error number:

[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_wait*() or *pthread_cond_timedwait*()] by another thread.
[EINVAL]	The value specified by *mutex* is invalid.

11.3.2.5 Cross-References

pthread_mutex_lock(), 11.3.3; *pthread_mutex_unlock*(), 11.3.3; *pthread_mutex_trylock*(), 11.3.3.

11.3.3 Locking and Unlocking a Mutex

Functions: *pthread_mutex_lock*(), *pthread_mutex_unlock*(), *pthread_mutex_trylock*().

11.3.3.1 Synopsis

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

11.3.3.2 Description

If {_POSIX_THREADS} is defined:

> The mutex object referenced by *mutex* shall be locked by calling *pthread_mutex_lock*(). If the mutex is already locked, the calling thread blocks until the mutex becomes available. This operation returns with the mutex object referenced by *mutex* in the locked state with the calling thread as its owner. An attempt by the current owner of a mutex to relock the mutex results in undefined behavior.
> The function *pthread_mutex_trylock*() is identical to *pthread_mutex_lock*() except that if the mutex object referenced by *mutex* is currently locked (by any thread, including the current thread), the call returns immediately.
> The function *pthread_mutex_unlock*() is called by the owner of the mutex object referenced by *mutex* to release it. A *pthread_mutex_unlock*() call by a thread that is not the owner of the mutex results in undefined behavior. Calling *pthread_mutex_unlock*() when the mutex object is unlocked also results in undefined behavior. If there are threads blocked on the mutex object referenced by *mutex* when *pthread_mutex_unlock*() is called, the *mutex* becomes available, and the scheduling policy is used to determine which thread shall acquire the mutex. See 13.6 for full details.
> If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the thread shall resume waiting for the mutex as if it was not interrupted.

Otherwise:

> Either the implementation shall support the *pthread_mutex_lock*(), *pthread_mutex_unlock*(), and *pthread_mutex_trylock*() functions as described above or the *pthread_mutex_lock*(), *pthread_mutex_unlock*(), and *pthread_mutex_trylock*() functions shall not be provided.

11.3.3.3 Returns

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 function *pthread_mutex_trylock*() shall return zero if a lock on the mutex object referenced by *mutex* is acquired. Otherwise, an error number shall be returned to indicate the error.

11.3.3.4 Errors

If any of the following conditions occur, the *pthread_mutex_lock*() and *pthread_mutex_trylock*() functions shall return the corresponding error number:

[EINVAL]	The *mutex* was created with the protocol attribute having the value PTHREAD_PRIO_PROTECT and the priority of the calling thread is higher than the current priority ceiling of the mutex. (See 13.6.)

If any of the following conditions occur, the *pthread_mutex_trylock*() function shall return the corresponding error number:

[EBUSY]	The *mutex* could not be acquired because it was already locked.

For each of the following conditions, if the condition is detected, the *pthread_mutex_lock*(), *pthread_mutex_trylock*(), and *pthread_mutex_unlock*() functions shall return the corresponding error number:

[EINVAL]	The value specified by *mutex* does not refer to an initialized mutex object.

For each of the following conditions, if the condition is detected, the *pthread_mutex_lock*() function shall return the corresponding error number:

[EDEADLK]	The current thread already owns the mutex.

For each of the following conditions, if the condition is detected, the *pthread_mutex_unlock*() function shall return the corresponding error number:

[EPERM]	The current thread does not own the mutex.

11.3.3.5 Cross-References

pthread_mutex_init(), 11.3.2; *pthread_mutex_destroy*(), 11.3.2.

11.4 Condition Variables

11.4.1 Condition Variable Initialization Attributes

Function: *pthread_condattr_init*(), *pthread_condattr_destroy*(), *pthread_condattr_getpshared*(), *pthread_condattr_setpshared*()

11.4.1.1 Synopsis

```
#include <pthread.h>
int pthread_condattr_init(pthread_condattr_t *attr);
int pthread_condattr_destroy(pthread_condattr_t *attr);
int pthread_condattr_getpshared(const pthread_condattr_t *attr,
            int *pshared);
int pthread_condattr_setpshared(pthread_condattr_t *attr,
            int pshared);
```

11.4.1.2 Description

If {_POSIX_THREADS} is defined:

The function *pthread_condattr_init*() initializes a condition variable attributes object *attr* with the default value for all of the attributes defined by the implementation.

Attempting to initialize an already initialized condition variable attributes object results in undefined behavior.

After a condition variable attributes object has been used to initialize one or more condition variables, any function affecting the attributes object (including destruction) does not affect any previously initialized condition variables.

The *pthread_condattr_destroy*() function destroys 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 condition variable attributes object can be reinitialized using *pthread_condattr_init*(); the results of otherwise referencing the object after it has been destroyed are undefined.

If the symbol {_POSIX_THREAD_PROCESS_SHARED} is defined, the implementation shall provide the attribute process-shared and the associated functions *pthread_condattr_getpshared*() and *pthread_condattr_setpshared*(). If this symbol is not defined, then the process-shared attribute and these functions are not supported. The process-shared attribute is set to PTHREAD_PROCESS_SHARED to permit a condition variable to be operated upon by any thread that has access to the memory where the condition variable is allocated, even if the condition variable is allocated in memory that is shared by multiple processes. If the process-shared attribute is PTHREAD_PROCESS_PRIVATE, the condition variable shall only be operated upon by threads created within the same process as the thread that initialized the condition variable; if threads of differing processes attempt to operate on such a condition variable, the behavior is undefined. The default value of the attribute shall be PTHREAD_PROCESS_PRIVATE.

The *pthread_condattr_setpshared*() function is used to set the process-shared attribute in an initialized attributes object referenced by *attr*. The *pthread_condattr_getpshared*() function obtains the value of the process-shared attribute from the attributes object referenced by *attr*.

Otherwise:

Either the implementation shall support the *pthread_condattr_init*(), *pthread_condattr_destroy*(), *pthread_condattr_getpshared*(), and *pthread_condattr_setpshared*() functions as described above or the *pthread_condattr_init*(), *pthread_condattr_destroy*(), *pthread_condattr_getpshared*(), and *pthread_condattr_setpshared*() functions shall not be provided.

See 13.4.1 for full details of attributes related to scheduling policies.

Additional attributes, their default values, and the names of the associated functions to get and set those attribute values are implementation defined.

11.4.1.3 Returns

If successful, the *pthread_condattr_init*(), *pthread_condattr_destroy*(), and *pthread_condattr_setpshared*() functions shall return zero. Otherwise, an error number shall be returned to indicate the error.

If successful, the *pthread_condattr_getpshared*() function shall return zero and store the value of the process-shared attribute of *attr* into the object referenced by the *pshared* parameter. Otherwise, an error number shall be returned to indicate the error.

11.4.1.4 Errors

If any of the following conditions occur, the *pthread_condattr_init*() function shall return the corresponding error number:

[ENOMEM] Insufficient memory exists to initialize the condition variable attributes object.

For each of the following conditions, if the condition is detected, the *pthread_condattr_destroy*(), *pthread_condattr_getpshared*(), and *pthread_condattr_setpshared*() functions shah return the corresponding error number:

[EINVAL] The value specified by *attr* is invalid.

For each of the following conditions, if the condition is detected, the *pthread_condattr_setpshared*() function shall return the corresponding error number:

[EINVAL] The new value specified for the attribute is outside the range of legal values for that attribute.

11.4.1.5 Cross-References

pthread_create(), 16.2.2; *pthread_mutex_init*(), 11.3.2; *pthread_cond_init*(), 11.4.2.

11.4.2 Initializing and Destroying Condition Variables

Functions: *pthread_cond_init*(), *pthread_cond_destroy*()

11.4.2.1 Synopsis

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *cond,
            const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

11.4.2.2 Description

If {_POSIX_THREADS} is defined:

The function *pthread_cond_init*() initializes the condition variable referenced by *cond* with attributes referenced by *attr*. If *attr* is **NULL**, the default condition variable attributes are 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 becomes initialized.
Attempting to initialize an already initialized condition variable results in undefined behavior.
The function *pthread_cond_destroy*() destroys 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.

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 the parameter *attr* specified as **NULL**, except that no error checks are performed.

Otherwise:

Either the implementation shall support the *pthread_cond_init*() and *pthread_cond_destroy*() functions as described above or the *pthread_cond_init*() and *pthread_cond_destroy*() functions shall not be provided.

11.4.2.3 Returns

If successful, the *pthread_cond_init*() and *pthread_cond_destroy*() 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*.

11.4.2.4 Errors

If any of the following conditions occur, the *pthread_cond_init*() function shall return the corresponding error number:

[EAGAIN]	The system lacked the necessary resources (other than memory) to initialize another condition variable.
[ENOMEM]	Insufficient memory exists to initialize the condition variable.

For each of the following conditions, if the condition is detected, the *pthread_cond_init*() function shall return the corresponding error number:

[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.

For each of the following conditions, if the condition is detected, the *pthread_cond_destroy*() function shall return the corresponding error number:

[EBUSY]	The implementation has detected an attempt to destroy the object referenced by *cond* while it is referenced by another thread [for example, while being used in a *pthread_cond_wait*() or a *pthread_cond_timedwait*()].
[EINVAL]	The value specified by *cond* is invalid.

11.4.2.5 Cross-References

pthread_cond_signal(), 11.4.3; *pthread_cond_broadcast*(), 11.4.3; *pthread_cond_wait*(), 11.4.4; *pthread_cond_timedwait*(), 11.4.4.

11.4.3 Broadcasting and Signaling a Condition

Functions: *pthread_cond_signal*(), *pthread_cond_broadcast*()

11.4.3.1 Synopsis

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

11.4.3.2 Description

If {_POSIX_THREADS} is defined:

These two functions are used to unblock threads blocked on a condition variable.
The *pthread_cond_signal*() call unblocks at least one of the threads that are blocked on the specified condition variable *cond* (if any threads are blocked on *cond*).
The *pthread_cond_broadcast*() call unblocks all threads currently blocked on the specified condition variable *cond*.
If more than one thread is blocked on a condition variable, the scheduling policy determines the order in which threads are unblocked. When each thread unblocked as a result of a *pthread_cond_signal*() or *pthread_cond_broadcast*() returns from its call to *pthread_cond_wait*() or *pthread_cond_timedwait*(), the thread owns the mutex with which it called *pthread_cond_wait*() or *pthread_cond_timedwait*(). The thread(s) that are unblocked shall contend for the mutex according to the scheduling policy (if applicable) and as if each had called *pthread_mutex_lock*(). See 13.6 for full details of the effects of the various scheduling policies.
The *pthread_cond_signal*() or *pthread_cond_broadcast*() functions may be called by a thread whether or not that thread currently owns the mutex that threads calling *pthread_cond_wait*() or *pthread_cond_timedwait*() have associated with the condition variable during their waits. However, if predictable scheduling behavior is required, then that mutex shall be locked by the thread calling *pthread_cond_signal*() or *pthread_cond_broadcast*().
The *pthread_cond_signal*() and *pthread_cond_broadcast*() functions have no effect if there are no threads currently blocked on *cond*.

Otherwise:

Either the implementation shall support the *pthread_cond_signal*() and *pthread_cond_broadcast*() functions as described above or the *pthread_cond_signal*() and *pthread_cond_broadcast*() functions shall not be provided.

11.4.3.3 Returns

If successful, the *pthread_cond_signal*() and *pthread_cond_broadcast*() functions shall return zero. Otherwise, an error number shall be returned to indicate the error.

11.4.3.4 Errors

For each of the following conditions, if the condition is detected, the *pthread_cond_signal*() and *pthread_cond_broadcast*() functions shall return the corresponding error number:

[EINVAL]	The value *cond* does not refer to an initialized condition variable.

11.4.3.5 Cross-References

pthread_cond_init(), 11.4.2; *pthread_cond_wait*(), 11.4.4; *pthread_cond_timedwait*(), 11.4.4.

11.4.4 Waiting on a Condition

Functions: *pthread_cond_wait*(), *pthread_cond_timedwait*()

11.4.4.1 Synopsis

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
            pthread_mutex_t *mutex, const struct timespec *abstime);
```

11.4.4.2 Description

If {_POSIX_THREADS} is defined:

The *pthread_cond_wait*() and *pthread_cond_timedwait*() functions are used to block on a condition variable. They shall be called with *mutex* locked by the calling thread or undefined behavior will result.

These functions release *mutex* and cause the calling thread to block on the condition variable *cond*. If another thread is able to acquire the mutex after the about-to-block thread has released it, then a subsequent call to *pthread_cond_signal*() or *pthread_cond_broadcast*() in that thread shall behave as if it were issued after the about-to-block thread has blocked.

Upon successful return, the mutex is locked and is owned by the calling thread.

When using condition variables, there is always a Boolean predicate involving shared variables associated with each condition wait that is true if the thread should proceed. Spurious wakeups from the *pthread_cond_wait*() or *pthread_cond_timedwait*() functions may occur. Since the return from *pthread_cond_wait*() or *pthread_cond_timedwait*() does not imply anything about the value of this predicate, the predicate should be re-evaluated upon each return.

The effect of using more than one mutex for concurrent *pthread_cond_wait*() or *pthread_cond_timedwait*() operations on the same condition variable is undefined—that is, a condition variable becomes bound to a unique mutex when a thread waits on the condition variable, and this (dynamic) binding ends when the wait returns.

A condition wait (whether timed or not) is a cancellation point. When the cancelability enable state of a thread is set to PTHREAD_CANCEL_DEFERRED, a side effect of acting upon a cancellation request while in a condition wait is that the mutex is (in effect) re-acquired before calling the first cancellation cleanup handler. The effect is as if the thread were unblocked, allowed to execute up to the point of returning from the call to *pthread_cond_wait*() or *pthread_cond_timedwait*(), but at that point notices the cancellation request and instead of returning to the caller of *pthread_cond_wait*() or *pthread_cond_timedwait*(), starts the thread cancellation activities, which includes calling cancellation cleanup handlers. See Section 18 for full details of cancellation and cancellation cleanup handlers.

A thread that has been unblocked because it has been canceled while blocked in a call to *pthread_cond_wait*() or *pthread_cond_timedwait*() 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 is the same as the *pthread_cond_wait*() function 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. When such timeouts occur, *pthread_cond_timedwait*() shall nonetheless release and reacquire the mutex referenced by *mutex*. The function *pthread_cond_timedwait*() 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 shall resume waiting for the condition variable as if it was not interrupted, or it shall return zero due to spurious wakeup.

Otherwise:

Either the implementation shall support the *pthread_cond_wait*() and *pthread_cond_timedwait*() functions as described above or the *pthread_cond_wait*() and *pthread_cond_timedwait*() functions shall not be provided.

11.4.4.3 Returns

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 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.

11.4.4.4 Errors

If any of the following conditions occur, the *pthread_cond_timedwait*() function shall return the corresponding error number:

[ETIMEDOUT]

The time specified by *abstime* to *pthread_cond_timedwait*() has passed.

For each of the following conditions, if the condition is detected, the *pthread_cond_wait*() and *pthread_cond_timedwait*() functions shall return the corresponding error number:

[EINVAL]	The value specified by *cond*, *mutex*, or *abstime* is invalid.
	Different mutexes were supplied for concurrent *pthread_cond_wait*() or *pthread_cond_timedwait*() operations on the same condition variable.
	The mutex was not owned by the current thread at the time of the call.

11.4.4.5 Cross-References

pthread_cond_signal(), 11.4.3; *pthread_cond_broadcast*(), 11.4.3.

12. Memory Management

This section describes the process memory locking, memory mapped files, and shared memory facilities available under this part of ISO/IEC 9945.

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

Memory locking guarantees the residence of portions of the address space. It is implementation defined whether locking memory guarantees fixed translation between virtual addresses (as seen by the process) and physical addresses. Per process memory locks are not inherited across a *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.

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

Implementations may support the Shared Memory Objects option without supporting a general Memory Mapped Files option. Shared memory objects are named regions of storage that may be independent of the file system and can be mapped into the address space of one or more processes to allow them to share the associated memory.

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

If the Memory Protection option is supported, the mapping may be restricted to disallow some types of access. When the Memory Protection option is supported, references to whole pages within the mapping but beyond the current length of an object shall result in a SIGBUS signal. SIGBUS is used in this context to indicate an error using the object. When the Memory Protection option is not supported, the result of references to memory within the mapping but beyond the current length of an object is undefined. The size of the object is unaffected by access beyond the end of the object. If the Memory Protection option is supported, write attempts to memory that was mapped without write access, or any access to memory mapped PROT_NONE, shall result in a SIGSEGV signal. SIGSEGV is used in this context to indicate, a mapping error. If the Memory Protection option is supported, references to unmapped addresses shall result in a SIGSEGV signal. If the Memory Protection option is not supported, the effect of references to unmapped addresses is undefined.

12.1 Memory Locking Functions

12.1.1 Lock/Unlock the Address Space of a Process

Functions: *mlockall*(), *munlockall*()

12.1.1.1 Synopsis

```
#include <sys/mman.h>
int mlockall(int flags);
int munlockall(void);
```

12.1.1.2 Description

If {_POSIX_MEMLOCK} is defined:

> The function *mlockall*() shall cause all of the pages mapped by the address space of a process to be memory resident until unlocked or until the process exits or *execs* another process image. The *flags* argument determines whether the pages to be locked are those currently mapped by the address space of the process, those that will be mapped in the future, or both. The *flags* argument is constructed from the 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 implementation defined.

The *munlockall*() function unlocks 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 are unaffected by a call by this process to *munlockall*().

Upon successful return from the *mlockall*() function that specifies MCL_CURRENT, all currently mapped pages of the process's address space shall be memory resident and locked. Upon return from the *munlockall*() function, all currently mapped pages of the process's address space shall be unlocked with respect to the process's address space. The memory residency of unlocked pages is unspecified.

The appropriate privilege is required to lock process memory with *mlockall*().

Otherwise:

Either the implementation shall support the *mlockall*() and *munlockall*() functions as described above or each of the *mlockall*() and *munlockall*() functions shall fail.

12.1.1.3 Returns

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.

12.1.1.4 Errors

If any of the following conditions occur, the *mlockall*() and *munlockall*() functions shall return −1 and set *errno* to the corresponding value:

[ENOSYS]	The implementation does not support this memory locking interface.

If any of the following conditions occur, the *mlockall*() function shall return −1 and set *errno* to the corresponding value:

[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.

For each of the following conditions, if the condition is detected, the *mlockall*() function shall return −1 and set *errno* to the corresponding value:

[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.

12.1.1.5 Cross-References

exec, 3.1.2; *_exit*(), 3.2.2; *fork*(), 3.1.1; *munmap*(), 12.2.2.

12.1.2 Lock/Unlock a Range of Process Address Space

Functions: *mlock*(), *munlock*()

12.1.2.1 Synopsis

```
#include <sys/mman.h>
int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);
```

12.1.2.2 Description

If {_POSIX_MEMLOCK_RANGE} is defined:

The function *mlock*() shall cause those whole pages containing any part of the address space of the process starting at address *addr* and continuing for *len* bytes to be memory resident until unlocked or until the process exits or *execs* another process image. The implementation may require that *addr* be a multiple of the page size {PAGESIZE}.

NOTE — It is expected that a later amendment of this standard will disallow the implementation from imposing the restriction on the alignment of *addr*.

The function *munlock*() 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 the page size {PAGESIZE}.

NOTE — It is expected that a later amendment of this standard will disallow the implementation from imposing the restriction on the alignment of *addr*.

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*().

Otherwise:

Either the implementation shall support the *mlock*() and *munlock*() functions as described above or each of the *mlock*() and *munlock*() functions shall fail.

12.1.2.3 Returns

Upon successful completion, the *mlock*() and *munlock*() functions shall return a value of zero. Otherwise, no change shall be 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.

12.1.2.4 Errors

If any of the following conditions occur, the *mlock*() and *munlock*() functions shall return −1 and set *errno* to the corresponding value:

[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.
[ENOSYS]	The implementation does not support this memory locking interface.

If any of the following conditions occur, the *mlock*() function shall return −1 and set *errno* to the corresponding value:

[EAGAIN]	Some or all of the memory identified by the operation could not be locked when the call was made.

For each of the following conditions, if the condition is detected, the *mlock*() and *munlock*() functions shall return −1 and set *errno* to the corresponding value:

[EINVAL]	The *addr* argument is not a multiple of the page size {PAGESIZE}.

For each of the following conditions, if the condition is detected, the *mlock*() function shall return −1 and set *errno* to the corresponding value:

[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 appropriate privilege to perform the requested operation.

12.1.2.5 Cross-References

exec, 3.1.2; *_exit*(), 3.2.2; *fork*(), 3.1.1; *munmap*(), 12.2.2.

12.2 Memory Mapping Functions

12.2.1 Map Process Addresses to a Memory Object

Function: *mmap*()

12.2.1.1 Synopsis

```
#include <sys/mman.h>
void *mmap(void *addr, size_t len, int prot, int flags, int fildes,
           off_t off);
```

12.2.1.2 Description

If at least one of {_POSIX_MAPPED_FILES} or {_POSIX_SHARED_MEMORY_OBJECTS} is defined:

The function *mmap*() establishes a mapping between the address space of the process and a memory object. The format of the call is as follows:

```
    pa = mmap(addr, len, prot, flags, fildes, off);
```

The *mmap*() function establishes 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-dependent function of the parameter *addr* and the values of *flags*, further described below. A successful *mmap*() call returns *pa* as its result. The address range starting at *pa* and continuing for *len* bytes shall be legitimate for the possible (not necessarily current) address space of the process. The range of bytes starting at *off* and continuing for *len* bytes shall be legitimate for the possible (not necessarily current) offsets in the file or shared memory object represented by *fildes*.

The mapping established by *mmap*() replaces 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.

The parameter *prot* determines whether read, write, execute, or some combination of accesses are permitted to the data being mapped. The *prot* should be either PROT_NONE or the bitwise inclusive OR of one or more of the other flags in Table 12.1, defined in the header `<sys/mman.h>`.

If an implementation cannot support the combination of access types specified by *prot*, the call to *mmap*() shall fail. An implementation may permit accesses other than those specified by *prot*; however, if the Memory Protection option is supported, the implementation shall not permit a write to succeed where PROT_WRITE has not been set or permit any access where

Table 12.1—Memory Protection Values

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.

PROT_NONE alone has been set. If the Memory Protection option is supported, the implementation shall support at least the following values of *prot*: PROT_NONE, PROT_READ, PROT_WRITE, and the inclusive OR of PROT_READ and PROT_WRITE. If the Memory Protection option is not supported, the result of any access that conflicts with the specified protection is undefined. The file descriptor *fildes* shall have been opened with read permission, regardless of the protection options specified. If PROT_WRITE is specified, the application shall have 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/roman.h>`:

Symbolic Constant	Description
MAP_SHARED	Changes are shared.
MAP_PRIVATE	Changes are private.
MAP_FIXED	Interpret *addr* exactly.

MAP_SHARED and MAP_PRIVATE describe the disposition of write references to the memory object. Implementations that do not support the Memory Mapped Files option are not required to support MAP_PRIVATE. If MAP_SHARED is specified, write references change the underling object. If the implementation supports MAP_PRIVATE and it 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 shall be specified, but not both. The mapping type is retained across *fork*().

MAP_FIXED informs the system that the value of *pa* shall be *addr* exactly. It is implementation defined whether MAP_FIXED is supported.

NOTE — For implementations that support MAP_FIXED, its use may result in poor performance.

When MAP_FIXED is not set, the system 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 system deems suitable for a mapping of *len* bytes to the specified object. An *addr* value of zero grants the system complete freedom in selecting *pa*, subject to the constraints described below. A nonzero value of *addr* is taken to be a suggestion of a process address near which the mapping should be placed. When the system selects a value for *pa*, it shall never place a mapping at address zero, nor shall it replace an extant mapping.

If MAP_FIXED is specified and *addr* is nonzero, it shall have the same remainder as the *off* parameter, modulo the page size {PAGESIZE}. 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.

NOTE — It is expected that a later amendment of this standard will disallow the implementation from imposing the restriction on the alignment of the *off* and *addr* arguments.

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 that are beyond its end. If the Memory Protection option is supported, 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 the generation of a SIGBUS signal. When the Memory Protection option is not supported, the result of references within the address range starting at *pa* and continuing for *len* bytes to whole pages following the end of an object is undefined.

An implementation may deliver SIGBUS signals when a reference would cause an error in the mapped object, such as out-of-space condition.

The *st_atime* field of the mapped memory object may be marked for update at any time between the *mmap*() call and the corresponding *mmunmap*() call. The initial read or write reference to a mapped region shall cause the *st_atime* field of the file to be marked for update if it has not already been marked for update.

The *st_ctime* and *st_mtime* fields of a memory object 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, these fields may be marked for update at any time after a write reference if the underlying file is modified as a result.

Otherwise:

Either the implementation shall support the *mmap*() function as described above or the *mmap*() function shall fail.

12.2.1.3 Returns

Upon successful completion, the *mmap*() function shall return the address at which the mapping was placed (*pa*); otherwise, it shall return a value of MAP_FAILED and set *errno* to indicate the error. The symbol MAP_FAILED shall be defined in the header `<sys/mman.h>`. No successful return from *mmap*() shall return the value MAP_FAILED.

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.

12.2.1.4 Errors

If any of the following conditions occur, the *mmap*() function shall return MAP_FAILED and set *errno* to the corresponding value:

[EACCES]	The file descriptor *fildes* is not open for read, regardless of the protection specified.
	The file descriptor *fildes* is not open for write and PROT_WRITE was specified for a MAP_SHARED type mapping.
[EAGAIN]	The mapping could not be locked in memory, if required by *mlockall*(), due to a lack of resources.
[EBADF]	The *fildes* argument is not a valid open file descriptor.
[EINVAL]	The value in *flags* is invalid (e.g., neither MAP_PRIVATE or MAP_SHARED).
[ENODEV]	The *fildes* argument refers to an object for which *mmap*() is meaningless, such as a terminal.
[ENOMEM]	MAP_FIXED was specified, and the address range starting at *addr* and continuing for *len* bytes exceeds that allowed for the address space of a process; or MAP_FIXED was not specified, and there is insufficient room in the address space to effect the mapping.
	The mapping could not be locked in memory, if required by *mlockall*(), because it would require more space than the system is able to supply.
[ENOSYS]	The function *mmap*() is not supported by this implementation.
[ENOTSUP]	MAP_FIXED or MAP_PRIVATE was specified in the *flags* argument, and the implementation does not support this functionality.
	The implementation does not support the combination of accesses requested in the *prot* argument.
[ENXIO]	The addresses in the range starting at *off* and continuing for *len* bytes are invalid for the object specified by *fildes*.
	MAP_FIXED was specified in *flags* and the combination of *addr*, *len*, and *off* is invalid for the object specified by *fildes*.

For each of the following conditions, if the condition is detected, the *mmap*() function shall return MAP_FAILED and set *errno* to the corresponding value:

[EINVAL]	The arguments *addr* (if MAP_FIXED was specified) or *off* are not multiples of the page size {PAGESIZE}.

12.2.1.5 Cross-References

open(), 5.3.1; *close*(), 6.3.1; *munmap*(), 12.2.2; *ftruncate*(), 5.6.7; *mlockall*(), 12.1.1; *shm_open*(), 12.3.1; `<sys/mman.h>`, 12.1.1.2.

12.2.2 Unmap Previously Mapped Addresses

Function: *munmap*()

12.2.2.1 Synopsis

```
#include <sys/mman.h>
int munmap(void *addr, size_t len);
```

12.2.2.2 Description

If at least one of {_POSIX_MAPPED_FILES} or {_POSIX_SHARED_MEMORY_OBJECTS} is defined:

The function *munmap*() removes any mappings for those entire pages containing any part of the address space of the process starting at *addr* and continuing for *len* bytes. Further references to these pages shall result in the generation of a SIGSEGV signal to the process. If there are no mappings in the specified address range, then *munmap*() shall have no effect.

The implementation may require that *addr* be a multiple of the page size, {PAGESIZE}.

NOTE — It is expected that a later amendment of this standard will disallow the implementation from imposing the restriction on the alignment of *addr*.

If a mapping to be removed was private, any modifications made in this address range shall be discarded.

Any memory locks (see 12.1.2 and 12.1.1) associated with this address range shall be removed, as if by an appropriate call to *munlock*().

The behavior of this function is unspecified if the mapping was not established by a call to *mmap*().

Otherwise:

Either the implementation shall support the *munmap*() function as described above or the *munmap*() function shall fail.

12.2.2.3 Returns

Upon successful completion, the *munmap*() function shall return a value of zero; otherwise, it shall return a value of −1 and set *errno* to indicate the error.

12.2.2.4 Errors

If any of the following conditions occur, the *munmap*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL]	Some of 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.
[ENOSYS]	The function *munmap*() is not supported by this implementation.

For each of the following conditions, if the condition is detected, the *munmap*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL]	The value of *addr* is not a multiple of the page size {PAGESIZE}.

12.2.2.5 Cross-References

mmap(), 12.2.1; *mlock*(), 12.1.2; *mlockall*(), 12.1.1; *munlock*(), 12.1.2; *unlink*(), 5.5.1; `<sys/mman.h>`, 12.1.1.2.

12.2.3 Change Memory Protection

Function: *mprotect*()

12.2.3.1 Synopsis

```
#include <sys/mman.h>
int mprotect(const void *addr, size_t len, int prot);
```

12.2.3.2 Description

If {_POSIX_MEMORY_PROTECTION} is defined:

The function *mprotect*() changes 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 the other values in Table 12.1.

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 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 inclusive OR of PROT_READ and PROT_WRITE. If PROT_WRITE is specified, the application shall have 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, {PAGESIZE}.

NOTE — It is expected that a later amendment of this standard will disallow the implementation from imposing the restriction on the alignment of *addr*.

The behavior of this function is unspecified if the mapping was not established by a call to *mmap*().

Otherwise:

Either the implementation shall support the *mprotect*() function as described above or the *mprotect*() function shall fail.

12.2.3.3 Returns

Upon successful completion, the *mprotect*() function shall return a value of zero; otherwise, it shall return a value of −1 and set *errno* to indicate the error.

If *mprotect*() fails for reasons other than [EINVAL], the protections on some of the pages in the address range starting at *addr* and continuing for *len* bytes may have been changed.

12.2.3.4 Errors

If any of the following conditions occur, the *mprotect*() function shall return −1 and set *errno* to the corresponding value:

[EACCES]	The memory object was not opened for read, regardless of the protection specified.
	The memory object was not opened for write, and PROT_WRITE was specified for a MAP_SHARED type mapping.
[EAGAIN]	The *prot* argument specifies PROT_WRITE on a MAP_PRIVATE mapping, and there are insufficient memory resources to reserve for locking the private pages, if required.
[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.
	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.
[ENOSYS]	The function *mprotect*() is not supported by this implementation.
[ENOTSUP]	The implementation does not support the combination of accesses requested in the *prot* argument.

For each of the following conditions, if the condition is detected, the *mprotect*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL]	The value of *addr* is not a multiple of the page size {PAGESIZE}.

12.2.3.5 Cross-References

mmap(), 12.2.1; *mlock*(), 12.1.2; *mlockall*(), 12.1.1; `<sys/mman.h>`, 12.1.1.2.

12.2.4 Memory Object Synchronization

Function: *msync*()

12.2.4.1 Synopsis

```
#include <sys/mman.h>
int msync(void *addr, size_t len, int flags);
```

12.2.4.2 Description

If {_POSIX_MAPPED_FILES} and {_POSIX_SYNCHRONIZED_IO} are defined:

The *msync*() function writes 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 then invalidates cached copies of data.
The implementation may require that *addr* be a multiple of the page size, {PAGESIZE}.

NOTE — It is expected that a later amendment of this standard will disallow the implementation from imposing the restriction on the alignment of *addr*.

For mappings to files, the *msync*() function shall assure that all write operations are completed as defined for synchronized I/O data integrity completion. It is unspecified whether the implementation also writes out other file attributes. When the *msync*() function is called on MAP_PRIVATE mappings, any modified data shall not be written to the underlying object and shall not cause such data to be made visible to other processes. It is unspecified whether data in MAP_PRIVATE mappings has any permanent storage locations. The effect of *msync*() on shared memory objects is unspecified.
The *flags* argument is constructed from the bitwise inclusive OR of one or more of the following flags defined in the header `<sys/mman.h>`:

Symbolic Constant	**Description**
MS_ASYNC	Perform asynchronous writes.
MS_SYNC	Perform synchronous writes.
MS_INVALIDATE	Invalidate cached data.

When MS_ASYNC is specified, *msync*() returns 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*() invalidates 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.
The behavior of this function is unspecified if the mapping was not established by a call to *mmap*().
If *msync*() causes any write to a file, the *st_crime* and *st_mtime* fields of the file shall be marked for update.

Otherwise:

> Either the implementation shall support the *msync*() function as described above or the *msync*() function shall fail.

12.2.4.3 Returns

Upon successful completion, the *msync*() function shall return a value of zero; otherwise, it shall return a value of –1 and set *errno* to indicate the error.

12.2.4.4 Errors

If any of the following conditions occur, the *msync*() function shall return –1 and set *errno* to the corresponding value:

[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.
[EINVAL]	The value in *flags* is invalid.
[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.
[ENOSYS]	The function *msync*() is not supported by this implementation.

For each of the following conditions, if the condition is detected, the *msync*() function shall return –1 and set *errno* to the corresponding value:

[EINVAL]	The value of *addr* is not a multiple of the page size {PAGESIZE}.

12.2.4.5 Cross-References

mmap(), 12.2.1; *mlock*(), 12.1.2; *mlockall*(), 12.1.1; `<sys/mman.h>`, 12.1.1.2.

12.3 Shared Memory Functions

12.3.1 Open a Shared Memory Object

Function: *shm_open*()

12.3.1.1 Synopsis

```
#include <sys/mman.h>
int shm_open(const char *name, int oflag, mode_t mode);
```

12.3.1.2 Description

If {_POSIX_SHARED_MEMORY_OBJECTS} is defined:

> The *shm_open*() function establishes a connection between a shared memory object and a file descriptor. It creates 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 shall conform to the construction rules for a pathname. If *name* begins with the slash character, then processes calling *shm_open*() with the same value of *name* shall refer to the same shared memory object, as long as that

name has not been removed. If *name* does not begin with the slash character, the effect is implementation defined. The interpretation of slash characters other than the leading slash character in *name* is implementation defined.

If successful, *shm_open*() returns 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 shall be set.

The file status flags and file access modes of the open file description shall be set according to the value of *oflag*. The *oflag* argument is the bitwise inclusive OR of the following flags defined in the header `<fcntl.h>`. Applications shall 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 shall have 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 shall be 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 for 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 shall have a size of zero.
O_EXCL	If O_EXCL and O_CREAT are set, *shm_open*() shall fail 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 shall be 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, shall persist 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.

Otherwise:

Either the implementation shall support the *shm_open*() function as described above or the *shm_open*() function shall fail.

12.3.1.3 Returns

Upon successful completion, the *shm_open*() function shall return a nonnegative integer representing the lowest numbered unused file descriptor. Otherwise, it shall return –1 and set *errno* to indicate the error.

12.3.1.4 Errors

If any of the following conditions occur, the *shm_open*() function shall return –1 and set *errno* to the corresponding value:

[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. The implementation shall document under what circumstances this error may be returned.
[EMFILE]	Too many file descriptors are currently in use by this process.
[ENAMETOOLONG]	The length of the *name* string exceeds {PATH_MAX}, or a path-name component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.
[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.
[ENOSYS]	The function *shm_open*() is not supported by this implementation.

12.3.1.5 Cross-References

close(), 6.3.1; *dup*(), 6.2.1; *exec*, 3.1.2; *fcntl*(), 6.5.2; `<fcntl.h>`, 6.5.1; *umask*(), 5.3.3; *shm_unlink*(), 12.3.2; *mmap*(), 12.2.1; `<sys/mman.h>`, 12.1.1.2.

12.3.2 Remove a Shared Memory Object

Function: *shm_unlink*()

12.3.2.1 Synopsis

```
int shm_unlink(const char *name);
```

12.3.2.2 Description

If {_POSIX_SHARED_MEMORY_OBJECTS} is defined:

> The *shm_unlink*() 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.

Otherwise:

> Either the implementation shall support the *shm_unlink*() function as described above or the *shm_unlink*() function shall fail.

12.3.2.3 Returns

Upon successful completion, a value of zero shall be returned. Otherwise, a value of –1 shall be returned and *errno* shall be set to indicate the error. If –1 is returned, the named shared memory object shall not be changed by this function call.

12.3.2.4 Errors

If any of the following conditions occur, the *shm_unlink*() function shall return −1 and set *errno* to the corresponding value:

[EACCES]	Permission is denied to unlink the named shared memory object.
[ENAMETOOLONG]	The length of the *name* string exceeds [NAME_MAX] while {POSIX_NO_TRUNC} is in effect.
[ENOENT]	The named shared memory object does not exist.
[ENOSYS]	The function *shm_unlink*() is not supported by this implementation.

12.3.2.5 Cross-References

shm_open(), 12.3.1; *close*(), 6.3.1; *mmap*(), 12.2.1; *munmap*(), 12.2.2.

13. Execution Scheduling

13.1 Scheduling Parameters

A scheduling parameter structure *sched_param* contains the scheduling parameters required for implementation of each scheduling policy supported. It is defined in `<sched.h>` and shall contain at least the following member:

Member Type	Member Name	Description
int	*sched_priority*	Process execution scheduling priority.

Implementations may add extensions as permitted in 1.3.1.1, item (2). Adding extensions to this structure, which may change the behavior of the application with respect to this standard when those fields in the structure are uninitialized, also requires that the extension be enabled as required by 1.3.1.1.

Inclusion of the `<sched.h>` header shall make visible the symbols allowed by this standard to be in the header `<time.h>`.

13.2 Scheduling Policies

The scheduling semantics described in this section 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 will result if the sequencing of other resources takes processor scheduling policy into account.

There is, conceptually, one thread list for each priority. Any runnable thread may be on any thread list. Multiple scheduling policies shall be provided. Each nonempty 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 (e.g., moving threads between and within lists).

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

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

Associated with each policy is a priority range. Each policy definition shall specify the minimum priority range for that policy. The priority ranges for each policy may or may 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 nonempty thread list to become a running thread, regardless of its associated policy. This thread is then removed from its thread list.

Three scheduling policies are specifically required; others may be implementation defined. The following symbols shall be defined in the header `<sched.h>`:

Symbol	**Description**
SCHED_FIFO	First in-first out (FIFO) scheduling policy.
SCHED_RR	Round robin scheduling policy.
SCHED_OTHER	Another scheduling policy.

The values of these symbols shall be distinct.

13.2.1 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. If the thread whose priority has been modified is a running thread or is runnable, runnable thread it then becomes the tail of the thread list for its new priority.
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) If a thread whose policy or priority has been modified is a running thread or is runnable, runnable thread it then becomes the tail of the thread list for its new priority.

7) When a running thread issues the *sched_yield*() function, the thread becomes the tail of the thread list for its priority.
8) At no other time shall the position of a thread with this scheduling policy within the thread lists be affected.

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

13.2.2 SCHED_RR

Conforming implementations shall include a scheduling policy called the round robin scheduling policy. This policy is 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 function *sched_rr_get_interval*() 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 will 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 shall complete the unexpired portion of its round-robin-interval time period.

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

13.2.3 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). Conforming implementations shall document the behavior of this policy as described in the definition of scheduling policy. The effect of scheduling threads with the SCHED_OTHER policy in a system in which other threads are executing under SCHED_FIFO or SCHED_RR shall thus be implementation defined. This policy is defined to allow strictly conforming applications to be able to indicate that they no longer need a realtime scheduling policy in a portable manner.

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

13.3 Process Scheduling Functions

13.3.1 Set Scheduling Parameters

Function: *sched_setparam*()

13.3.1.1 Synopsis

```
#include <sched.h>
int sched_setparam(pid_t pid, const struct sched_param *param);
```

13.3.1.2 Description

If {_POSIX_PRIORITY_SCHEDULING} is defined:

> The *sched_setparam*() function sets 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 *param* structure shall be any integer within the inclusive priority range for the current scheduling policy of the process specified by *pid*. Higher numerical values for the priority represent higher priorities. If the value of *pid* is negative, the behavior of the *sched_setparam*() function is unspecified.
> If a process specified by *pid* exists and if the calling process has permission, the scheduling parameters shall be set for the process whose process ID is equal to *pid*.
> If *pid* is zero, the scheduling parameters shall be set for the calling process.
> The conditions under which one process has permission to change the scheduling parameters of another process are implementation defined.
> Implementations may require the requesting process to have the appropriate privilege to set its own scheduling parameters or those of another process.
> The target process, whether it is running or not running, shall resume execution after all other runnable processes of equal or greater priority have been scheduled to run.
> If the priority of the process specified by the *pid* argument is set higher than that of the lowest priority running process and if the specified process is ready to run, the process specified by the *pid* argument shall preempt a lowest priority running process. Similarly, if the process calling *sched_setparam*() sets its own priority lower than that of one or more other nonempty process lists, then the process that is the head of the highest priority list shall also preempt the calling process. Thus, in either case, the originating process might not receive notification of the completion of the requested priority change until the higher priority process has executed.
> If the current scheduling policy for the process specified by *pid* is not SCHED_FIFO or SCHED_RR, including SCHED_OTHER, the result is implementation defined.
> The effect of this function on individual threads is dependent on the scheduling contention scope of the threads. For threads with system scheduling contention scope, these functions shall have no effect on their scheduling. For threads with process scheduling contention scope, scheduling with respect to threads in other processes may be dependent on the scheduling policy and scheduling parameters of their particular process, which is governed using these functions.

Otherwise:

> Either the implementation shall support the *sched_setparam*() function as described above or the *sched_setparam*() function shall fail.

13.3.1.3 Returns

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.

13.3.1.4 Errors

If any of the following conditions occur, the *sched_setparam*() function shall return −1 and set *errno*to the corresponding value:

[EINVAL]	One or more of the requested scheduling parameters is outside the range defined for the scheduling policy of the specified *pid*.
[ENOSYS]	The function *sched_setparam*() is not supported by this implementation.

[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*.

13.3.1.5 Cross-References

sched_getparam(), 13.3.2; *sched_getscheduler*(), 13.3.4; *sched_setscheduler*(), 13.3.3.

13.3.2 Get Scheduling Parameters

Function: *sched_getparam*()

13.3.2.1 Synopsis

```
#include <sched.h>
int sched_getparam(pid_t pid, struct sched_param *param);
```

13.3.2.2 Description

If {_POSIX_PRIORITY SCHEDULING} is defined:

> The *sched_getparam*() function shall return the scheduling parameters of a process specified by *pid* in the *sched_param* structure pointed to by *param*.
> If a process specified by *pid* exists and if the calling process has permission, the scheduling parameters for the process whose process ID is equal to *pid* shall be returned.
> If *pid* is zero, the scheduling parameters for the calling process shall be returned. The behavior of the *sched_getparam*() function is unspecified if the value of *pid* is negative.

Otherwise:

> Either the implementation shall support the *sched_getparam*() function as described above or the *sched_getparam*() function shall fail.

13.3.2.3 Returns

Upon successful completion, the *sched_getparam*() function shall return zero. If the call to *sched_getparam*() is unsuccessful, the function shall return a value of −1 and set *errno* to indicate the error.

13.3.2.4 Errors

If any of the following conditions occur, the *sched_getparam*() function shall return −1 and set *errno* to the corresponding value:

[ENOSYS]	The function *sched_getparam*() is not supported by this implementation.
[EPERM]	The requesting process does not have permission to obtain the scheduling parameters of the specified process.
[ESRCH]	No process can be found corresponding to that specified by *pid*.

13.3.2.5 Cross-References

sched_getscheduler(), 13.3.4; *sched_setparam*(), 13.3.1; *sched_setscheduler*(), 13.3.3.

13.3.3 Set Scheduling Policy and Scheduling Parameters

Function: *sched_setscheduler*()

13.3.3.1 Synopsis

```
#include <sched.h>
int sched_setscheduler(pid_t pid, int policy,
            const struct sched_param *param);
```

13.3.3.2 Description

If {_POSIX_PRIORITY_SCHEDULING} is defined:

> The *sched_setscheduler*() function sets 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 *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 header file `<sched.h>`.
> If a process specified by *pid* exists and if the calling process has permission, the scheduling policy and scheduling parameters shall be set for the process whose process ID is equal to *pid*.
> If *pid* is zero, the scheduling policy and scheduling parameters shall be set for the calling process.
> The conditions under which one process has the appropriate privilege to change the scheduling parameters of another process are implementation defined.
> Implementations may require that the requesting process have permission to set its own scheduling parameters or those of another process. Additionally, implementation-defined restrictions may apply as to the appropriate privileges required to set a process's own scheduling policy, or another process's scheduling policy, to a particular value.
> The *sched_setscheduler*() function shall be considered successful if it succeeds in setting the scheduling policy and scheduling parameters of the process specified by *pid* to the values specified by *policy* and the structure *param*, respectively.
> The effect of this function on individual threads is dependent on the scheduling contention scope of the threads. For threads with system scheduling contention scope, these functions shall have no effect on their scheduling. For threads with process scheduling contention scope, scheduling with respect to threads in other processes may be dependent on the scheduling policy and scheduling parameters of their particular process, which is governed using these functions.

Otherwise:

> Either the implementation shall support the *sched_setscheduler*() function as described above or the *sched_setscheduler*() function shall fail.

13.3.3.3 Returns

Upon successful completion, the function shall return the former scheduling policy of the specified process. If the *sched_setscheduler*() function fails to complete successfully, the policy and scheduling paramenters shall remain unchanged, and the function shall return a value of –1 and set *errno* to indicate the error.

13.3.3.4 Errors

If any of the following conditions occur, the *sched_setscheduler*() function shall return –1 and set *errno* to the corresponding value:

[EINVAL]	The value of the *policy* parameter is invalid, or one or more of the parameters contained in *param* is outside the valid range for the specified scheduling policy.
[ENOSYS]	The function *sched_setscheduler*() is not supported by this implementation.
[EPERM]	The requesting process does not have permission to set either or both of the scheduling parameters or the scheduling policy of the specified process.
[ESRCH]	No process can be found corresponding to that specified by *pid*.

13.3.3.5 Cross-References

sched_getparam(), 13.3.2; *sched_getscheduler*(), 13.3.4; *sched_setparam*(), 13.3.1.

13.3.4 Get Scheduling Policy

Function: *sched_getscheduler*()

13.3.4.1 Synopsis

```
#include <sched.h>
int sched_getscheduler(pid_t pid);
```

13.3.4.2 Description

If {_POSIX_PRIORITY_SCHEDULING} is defined:

> The *sched_getscheduler*() function shall return the scheduling policy of the process specified by *pid*. If the value of *pid* is negative, the behavior of the *sched_getscheduler*() function is unspecified.
> The values that can be returned by *sched_getscheduler*() are defined in the header file `<sched.h>` [see *sched_setscheduler*()].
> If a process specified by *pid* exists and if the calling process has permission, the scheduling policy shall be returned for the process whose process ID is equal to *pid*.
> If *pid* is zero, the scheduling policy shall be returned for the calling process.

Otherwise:

> Either the implementation shall support the *sched_getscheduler*() function as described above or the *sched_getscheduler*() function shall fail.

13.3.4.3 Returns

Upon successful completion, the *sched_getscheduler*() function shall return the scheduling policy of the specified process. If unsuccessful, the function shall return –1 and set *errno* to indicate the error.

13.3.4.4 Errors

If any of the following conditions occur, the *sched_getscheduler*() function shall return –1 and set *errno* to the corresponding value:

[ENOSYS]	The function *sched_getscheduler*() is not supported by this implementation.
[EPERM]	The requesting process does not have permission to determine the scheduling policy of the specified process.
[ESRCH]	No process can be found corresponding to that specified by *pid*.

13.3.4.5 Cross-References

sched_getparam(), 13.3.2; *sched_setparam*(), 13.3.1; *sched_setscheduler*(), 13.3.3.

13.3.5 Yield Processor

Function: *sched_yield*()

13.3.5.1 Synopsis

```
#include <sched.h>
int sched_yield(void);
```

13.3.5.2 Description

If at least one of {_POSIX_PRIORITY_SCHEDULING} or {_POSIX_THREADS} is defined:

> The *sched_yield*() function forces the running thread to relinquish the processor until it again becomes the head of its thread list.

Otherwise:

> Either the implementation shall support the *sched_yield*() function as described above or the *sched_yield*() function shall fail.

13.3.5.3 Returns

The *sched_yield*() function shall return zero if it completes successfully, or it shall return a value of −1 and set *errno* to indicate the error.

13.3.5.4 Errors

If any of the following conditions occur, the *sched_yield*() function shall return −1 and set *errno* to the corresponding value:

[ENOSYS]	The *sched_yield*() function is not supported by this implementation.

13.3.6 Get Scheduling Parameter Limits

Functions: *sched_get priority_max*(), *sched_get_priority_min*(), *sched_rr_get_interval*()

13.3.6.1 Synopsis

```
#include <sched.h>
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
int sched_rr_get_interval(pid_t pid, struct timespec *interval);
```

13.3.6.2 Description

If {_POSIX_PRIORITY_SCHEDULING} is defined:

> The *sched_get_priority_max*() and *sched_get_priority_min*() functions return the appropriate maximum or minimum, respectfully, for the scheduling policy specified by *policy*. The *sched_rr_get_interval*() function

updates the timespec structure referenced by the interval argument to contain the current execution time limit (i.e., time quantum) for the process specified by *pid*. If *pid* is zero, the current execution time limit for the calling process shall be returned.
The value of *policy* shall be one of the scheduling policy values defined in `<sched.h>`.

Otherwise:

Either the implementation shall support the *sched_get_priority_max*(), *sched_get_priority_min*(), and *sched_rr_get_interval*() functions as described above or each of the *sched_get_priority_max*(), *sched_get_priority_min*(), and *sched_rr_get_interval*() functions shall fail.

13.3.6.3 Returns

If successful, the *sched_get_priority_max*() and *sched_get_priority_min*() functions shall return the appropriate maximum or minimum values, respectively. If unsuccessful, they shall return a value of −1 and set *errno* to indicate the error.

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.

13.3.6.4 Errors

If any of the following conditions occur, the *sched_get_priority_max*(), *sched_get_priority_min*(), and *sched_rr_get_interval*() functions shall return −1 and set *errno* to the corresponding value:

[EINVAL]	The value of the *policy* parameter does not represent a defined scheduling policy.
[ENOSYS]	The *sched_get_priority_max*(), *sched_get_priority_min*(), and *sched_rr_get_interval*() functions are not supported by this implementation.
[ESRCH]	No process can be found corresponding to that specified by *pid*.

13.3.6.5 Cross-References

sched_getparam(), 13.3.2; *sched_setparam*(), 13.3.1; *sched_getscheduler*(), 13.3.4; *sched_setscheduler*(), 13.3.3.

13.4 Thread Scheduling

This clause defines a set of operations that provide an extensible interface for the programmed control of scheduling of multiple threads within a single POSIX.1 process and within a POSIX.1 system. In this section, the scheduling interface is separate from the scheduling policies, and some of the functions are to be provided by any implementation that supports threads. However, implementations are not required to support any particular scheduling policies. Particular scheduling policies are specified under other options, including a set of priority based policies. The support of scheduling policies, including the optional priority-based policies defined by this standard, includes support for scheduling related attributes and functions of the synchronization mechanisms described in Section 11 These scheduling-related synchronization attributes and functions are described in this section. Thus, an implementation may support the scheduling interface without necessarily supporting the particular scheduling policies described or the scheduling policy-dependent synchronization attributes and functions.

13.4.1 Thread Scheduling Attributes

In support of the scheduling interface, threads have attributes that are accessed through the *pthread_attr_t* thread creation attributes object.

The contention scope 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 shall be described as part of the definition of those policies.

If the {_POSIX_THREAD_PRIORITY_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 shall ensure 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 as defined in 13.2.

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.

13.4.2 Scheduling Contention Scope

The scheduling contention scope of a thread defines the set of threads with which the thread has to compete for use of the processing resources. The scheduling operation will select at most one thread to execute on each processor at any point in time and the scheduling attributes of the thread (e.g., 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, effects threads as follows:

- A thread created with the 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 the 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 the PTHREAD_SCOPE_PROCESS scheduling contention scope are the implementation-defined mappings into the system attribute space of the scheduling attributes with which the thread was created.
- Threads created with the PTHREAD_SCOPE_PROCESS scheduling contention scope contend directly with other threads within their process that were created with the PTHREAD_SCOPE_PROCESS scheduling contention scope. The contention is resolved based on the scheduling attributes and policies of the threads. It is unspecified how such threads are scheduled relative to threads in other processes or threads with the 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.

13.4.3 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 multiprocessors may map all or any subset of the CPUs to one or multiple scheduling allocation domains. They 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 has to contend 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 shall be 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 whose scheduling allocation domain size is equal to one, the scheduling rules defined for SCHED_FIFO and SCHED_RR in 13.2 shall be used. 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 whose scheduling allocation domain size is greater than one, the rules defined for SCHED_FIFO and SCHED-RR in 13.2 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.

13.4.4 Scheduling Documentation

If {_POSIX_THREAD_PRIORITY_SCHEDULING} is defined, then any scheduling policies beyond SCHED_OTHER, SCHED_FIFO, and SCHED_RR, 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.

13.5 Thread Scheduling Functions

13.5.1 Thread Creation Scheduling Attributes

Functions: *pthread_attr_setscope*(), *pthread_attr_getscope*(), *pthread_attr_setinheritsched*(), *pthread_attr_-getinheritsched*(), *pthread_attr_setschedpolicy*(), *pthread_attr_getschedpolicy*(), *pthread_attr_setschedparam*(), *pthread_attr_getschedparam*()

13.5.1.1 Synopsis

```
#include <pthread.h>
int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
int pthread_attr_getscope(const pthread_attr_t *attr,
            int *contentionscope);
int pthread_attr_setinheritsched(pthread_attr_t *attr,
            int inheritsched);
int pthread_attr_getinheritsched(const pthread_attr_t *attr,
            int *inheritsched);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy(const pthread_attr_t *attr,
            int *policy);
int pthread_attr_setschedparam(pthread_attr_t *attr,
            const struct sched_param *param);
int pthread_attr_getschedparam(const pthread_attr_t *attr,
            struct sched_param *param);
```

13.5.1.2 Description

If {_POSIX_THREAD_PRIORITY_SCHEDULING} is defined:

The *pthread_attr_setscope*() and *pthread_attr_getscope*() functions are used to set and get 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 shall be defined by the header `<pthread.h>`.

The functions *pthread_attr_setinheritsched*() and *pthread_attr_ getinheritsched*() set and get the `inheritsched` attribute in the *attr* argument.

When the thread attributes objects are used by *pthread_create*(), the `inheritsched` attribute determines how the other scheduling attributes of the created thread are to be set:

PTHREAD_INHERIT_SCHEDSpecifies that the scheduling policy and associated attributes are to be inherited from the creating thread, and the scheduling attributes in this *attr* argument are to be ignored.

PTHREAD_EXPLICIT_SCHEDSpecifies that the scheduling policy and associated attributes are to be set to the corresponding values from this attributes object.

The symbols PTHREAD_INHERIT_SCHED and PTHREAD_EXPLICIT_SCHED shall be defined in the header `<pthread.h>`.

The functions *pthread_attr_setschedpolicy*() and *pthread_attr_getschedpolicy*() set and get the `schedpolicy` attribute in the *attr* argument.

The supported values of *policy* shall include SCHED_FIFO, SCHED_RR, or SCHED_OTHER, which shall be defined by the header `<sched.h>`. The meaning of the value of *policy*, for these values of `schedpolicy`, is the same as that defined in 13.2. When threads executing with the scheduling policy SCHED_FIFO or SCHED_RR are waiting on a mutex, they will acquire the mutex in priority order when the mutex is unlocked. See 11.3 for details of mutex handling.

Otherwise:

Either the implementation shall support the *pthread_attr_setscope*(), *pthread_attr_getscope*(), *pthread_attr_setinheritsched*(), *pthread_attr_getinheritsched*(), *pthread_attr_setschedpolicy*(), and *pthread_attr_getschedpolicy*() functions as described above or the *pthread_attr_setscope*(), *pthread_attr_getscope*(), *pthread_attr_setinheritsched*(), *pthread_attr_getinheritsched*(), *pthread_attr_setschedpolicy*(), and *pthread_attr_getschedpolicy*() functions shall not be provided.

If {_POSIX_THREADS} is defined:

The functions *pthread_attr_setschedparam*() and *pthread_attr_getschedparam*() set and get the `schedparam` attribute in the *attr* argument. The contents of the *param* structure are defined in 13.1. For the SCHED_FIFO and SCHED_RR policies, the only required member of the *param* is *sched_priority*.

Otherwise:

Either the implementation shall support the *pthread_attr_setschedparam*() and *pthread_attr_getschedparam*() functions as described above or the *pthread_attr_setschedparam*() and *pthread_attr_getschedparam*() functions shall not be provided.

13.5.1.3 Returns

If successful, the *pthread_attr_setscope*(), *pthread_attr_getscope*(), *pthread_attr_setinheritsched*(), *pthread_attr_-getinheritsched*(), *pthread_attr_setschedpolicy*(), *pthread_attr_getschedpolicy*(), *pthread_attr_setschedparam*(), and *pthread_attr_getschedparam*() functions shall return zero. Otherwise, an error number shall be returned to indicate the error.

13.5.1.4 Errors

If any of the following conditions occur, the *pthread_attr_setscope*(), *pthread_attr_getscope*(), *pthread_attr_setinheritsched*(), *pthread_attr_getinheritsched*(), *pthread_attr_setschedpolicy*(), *pthread_attr_-getschedpolicy*(), *pthread_attr_setschedparam*(), and *pthread_attr_getschedparam*() functions shall return the corresponding error number:

[ENOSYS] The implementation does not support the *pthread_attr_setscope*(), *pthread_attr_getscope*(), *pthread_attr_setinheritsched*(), *pthread_attr_getinheritsched*(), *pthread_attr_setschedpolicy*(), *pthread_attr_getschedpolicy*(), *pthread_attr_setschedparam*(), and *pthread_attr_getschedparam*() functions.

For each of the following conditions, if the condition is detected, the *pthread_attr_setscope*(), *pthread_attr_setinheritsched*(), *pthread_attr_setschedpolicy*(), and *pthread_attr_setschedparam*() functions shall return the corresponding error number:

[EINVAL] The value of the attribute being set is not valid.

[ENOTSUP] An attempt was made to set the attribute to an unsupported value.

13.5.1.5 Cross-References

pthread_attr_init(), 16.2.1; *pthread_create*(), 16.2.2.

13.5.2 Dynamic Thread Scheduling Parameters Access

Functions: *pthread_getschedparam*(), *pthread_setschedparam*()

13.5.2.1 Synopsis

```
#include <pthread.h>
int pthread_getschedparam(pthread_t thread, int *policy,
            struct sched_param *param);
int pthread_setschedparam(pthread_t thread, int policy,
            const struct sched_param *param);
```

13.5.2.2 Description

If {_POSIX_THREAD_PRIORITY_SCHEDULING} is defined:

> The *pthread_getschedparam*() and *pthread_setschedparam*() functions allow the scheduling policy and scheduling parameters of individual threads within a multithreaded 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 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*() 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, *sched_priority*.
If the *pthread_setschedparam*() fails, no scheduling parameters shall be changed for the target thread.

Otherwise:

Either the implementation shall support the *pthread_getschedparam*() and *pthread_setschedparam*() functions as described above or the *pthread_getschedparam*() and *pthread_setschedparam*() functions shall not be provided.

13.5.2.3 Returns

If successful, the *pthread_getschedparam*() and *pthread_setschedparam*() functions shall return zero. Otherwise, an error number shall be returned to indicate the error.

13.5.2.4 Errors

If any of the following conditions occur, the *pthread_getschedparam*() and *pthread_setschedparam*() functions shall return the corresponding error number:

[ENOSYS]	The option {_POSIX_THREAD_PRIORITY_SCHEDULING} is not defined and the implementation does not support the *pthread_getschedparam*() and *pthread_setschedparam*() functions.

For each of the following conditions, if the condition is detected, the *pthread_getschedparam*() function shall return the corresponding error number:

[ESRCH]	The value specified by *thread* does not refer to a existing thread.

For each of the following conditions, if the condition is detected, the *pthread_setschedparam*() function shall return the corresponding error number:

[EINVAL]	The value specified by *policy* or one of the scheduling parameters associated with the scheduling policy *policy* is invalid.
[ENOTSUP]	An attempt was made to set the policy or scheduling parameters to an unsupported value.
[EPERM]	The caller does not have the appropriate permission to set either the scheduling parameters or the scheduling policy of the specified thread.
	The implementation does not allow the application to modify one of the parameters to the value specified.
[ESRCH]	The value specified by *thread* does not refer to a existing thread.

13.5.2.5 Cross-References

sched_setparam(), 13.3.1; *sched_getparam*(), 13.3.2; *sched_setscheduler*(), 13.3.3; *sched_getscheduler*(), 13.3.4.

13.6 Synchronization Scheduling

13.6.1 Mutex Initialization Scheduling Attributes

Functions: *pthread_mutexattr_setprotocol*(), *pthread_mutexattr_getprotocol*(), *pthread_mutexattr_setprioceiling*(), *pthread mutexattr_getprioceiling*()

13.6.1.1 Synopsis

```
#include <pthread.h>
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
            int protocol);
int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *attr,
            int *protocol);
int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
            int prioceiling);
int pthread_mutexattr_getprioceiling (
            const pthread_mutexattr_t *attr, int *prioceiling);
```

13.6.1.2 Description

If at least one of {_POSIX_THREAD_PRIO_INHERIT} or {_POSIX_THREAD_PRIO_PROTECT} is defined:

These functions shall manipulate a mutex attributes object pointed to by *attr*, which has been previously created by the function *pthread_mutexattr_init*() (see 11.3.1).

The *pthread_mutexattr_t* mutex attributes object shall include at least the `protocol` attribute.

If the symbol {_POSIX_THREAD_PRIO_PROTECT} is defined, the *pthread_mutexattr_t* mutex attributes object shall include the `prioceiling` attribute.

The `prioceiling` attribute contains the priority ceiling of initialized mutexes. The values of *prioceiling* shall be within the maximum range of priorities defined by SCHED_FIFO.

The `protocol` attribute defines the protocol to be followed in utilizing mutexes. The value of *protocol* may be one of PTHREAD_PRIO_NONE, PTHREAD_PRIO_INHERIT, or PTHREAD_PRIO_PROTECT, which shall be defined by the header `<pthread.h>`. The PTHREAD_PRIO_PROTECT value shall be valid if the symbol {_POSIX_THREAD_PRIO_PROTECT} is defined, and the PTHREAD_PRIO_INHERIT value shall be valid if the symbol {_POSIX_THREAD_PRIO_INHERIT} is defined.

When a thread owns a mutex with the PTHREAD_PRIO_NONE protocol attribute, its priority and scheduling are not affected by its mutex ownership.

When a thread is blocking higher priority threads because of owning one or more mutexes with the PTHREAD_PRIO_INHERIT protocol attribute, it executes at the higher of either its priority or the priority of the highest priority thread waiting on any of the mutexes owned by this thread and initialized with this protocol.

When a thread owns one or more mutexes initialized with the PTHREAD_PRIO_PROTECT protocol, it executes at the higher of either its priority or the highest of the priority ceilings of all the mutexes owned by this thread and initialized with this attribute, regardless of whether or not other threads are blocked on any of these mutexes. The `prioceiling` attribute defines the priority ceiling of initialized mutexes, which is the minimum priority level at which the critical section guarded by the mutex is executed. In order to avoid priority inversion, the priority ceiling of the mutex shall be set to a priority higher than or equal to the highest priority of all the threads that may lock that mutex. The values of `prioceiling` shall be within the maximum range of priorities defined under the SCHED_FIFO scheduling policy.

While a thread is holding a mutex that has been initialized with the PRIO_INHERIT or 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 PRIO_INHERIT or 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.

If the symbol {_POSIX_THREAD_PRIO_INHERIT} is defined, when a thread makes a call to *pthread_mutex_lock*() on a mutex that was initialized with the protocol attribute having the value PTHREAD_PRIO_INHERIT and 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.

Otherwise:

Either the implementation shall support the *pthread_mutexattr_setprotocol*(), *pthread_mutexattr_-getprotocol*(), *pthread_mutexattr_setprioceiling*(), and *pthread_mutexattr_getprioceiling*() functions as described above or the *pthread_mutexattr_setprotocol*(), *pthread_mutexattr_getprotocol*(), *pthread_-mutexattr_setprioceiling*(), and *pthread_mutexattr_getprioceiling*() functions shall not be provided.

13.6.1.3 Returns

Upon successful completion, the *pthread_mutexattr_setprotocol*(), *pthread_mutexattr_getprotocol*(), *pthread_mutexattr_setprioceiling*(), and *pthread_mutexattr_getprioceiling*() functions shall return zero. Otherwise, number shall be returned to indicate the error.

13.6.1.4 Errors

If any of the following conditions occur, the *pthread_mutexattr_setprotocol*(), *pthread_mutexattr_getprotocol*(), *pthread_mutexattr_setprioceiling*(), and *pthread_mutexattr_getprioceiling*() functions shall return the corresponding error number:

[ENOSYS]	The option {_POSIX_THREAD_PRIO_PROTECT} is not defined, and the implementation does not support the *pthread_mutexattr_setprioceiling*() and *pthread_mutexattr_getprioceiling*() functions.
[ENOTSUP]	The value specified by *protocol* is an unsupported value.

For each of the following conditions, if the condition is detected, the *pthread_mutexattr_setprotocol*(), *pthread_mutexattr_getprotocol*(), *pthread_mutexattr_setprioceiling*(), and *pthread_mutexattr_getprioceiling*() functions shall return the corresponding error number:

[EINVAL]	The value specified by *attr*, *protocol*, or *prioceiling* is invalid.
[EPERM]	The caller does not have the privilege to perform the operation.

13.6.1.5 Cross-References

pthread_create(), 16.2.2; *pthread_mutex_init*(), 11.3.2; *pthread_cond_init*(), 11.4.2.

13.6.2 Change the Priority Ceiling of a Mutex

Functions: *pthread_mutex_getprioceiling*(), *pthread_mutex_setprioceiling*()

13.6.2.1 Synopsis

```
#include <pthread.h>
int pthread_mutex_setprioceiling(pthread_mutex_t *mutex,
            int prioceiling, int *old_ceiling);
int pthread_mutex_getprioceiling(const pthread_mutex_t *mutex,
            int *prioceiling);
```

13.6.2.2 Description

If {_POSIX_THREAD_PRIO_PROTECT} is defined:

> The *pthread_mutex_getprioceiling*() function returns the current priority ceiling of the mutex.
> The *pthread_mutex_setprioceiling*() function either locks the mutex if it is unlocked, or blocks until it can successfully lock the mutex; then it changes the priority ceiling of the mutex and releases the mutex. When the change is successful, the previous value of the priority ceiling is returned in *old_ceiling*. The process of locking the mutex need not adhere to the priority protect protocol.
> If the *pthread_mutex_setprioceiling*() function fails, the mutex priority ceiling is not changed.

Otherwise:

> Either the implementation shall support the *pthread_mutex_getprioceiling*() and *pthread_mutex_setprioceiling*() functions as described above or the *pthread_mutex_getprioceiling*() and *pthread_mutex_setprioceiling*() functions shall not be provided.

13.6.2.3 Returns

If successful, the *pthread_mutex_setprioceiling*() and *pthread_mutex_getprioceiling*() functions shall return zero. Otherwise, an error number shall be returned to indicate the error.

13.6.2.4 Errors

If any of the following conditions occur, the *pthread_mutex_getprioceiling*() and *pthread_mutex_setprioceiling*() functions shall return the corresponding error number:

[ENOSYS]	The option {_POSIX_THREAD_PRIO_PROTECT} is not defined, and the implementation does not support the *pthread_mutex_setprioceiling*() and *pthread_mutex_getprioceiling*() functions.

For each of the following conditions, if the condition is detected, the *pthread_mutex_setprioceiling*() and *pthread_mutex_getprioceiling*() functions shall return the corresponding error number:

[EINVAL]	The priority requested by *prioceiling* is out of range.
	The value specified by *mutex* does not refer to a currently existing mutex.
[ENOSYS]	The implementation does not support the priority ceiling protocol for mutexes.
[EPERM]	The caller does not have the privilege to perform the operation.

13.6.2.5 Cross-References

pthread_mutex_init(), 11.3.2; *pthread_mutex_lock*() 11.3.3; *thread_mutex_unlock*(), 11.3.3; *pthread_mutex_trylock*(), 11.3.3.

14. Clocks and Timers

14.1 Data Definitions for Clocks and Timers

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

14.1.1 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

Implementations may add extensions as permitted in 1.3.1.1, item (2). Adding extensions to this structure, which may change the behavior of the application with respect to this standard when those fields in the structure are uninitialized, also requires that the extension be enabled as required by 1.3.1.1.

The *tv_nsec* member is only valid if greater than or equal to zero, and less than the number of nanoseconds in a second (1000 million). The time interval described by this structure is ($tv\_sec \times 10^9 + tv\_nsec$) 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

Implementations may add extensions as permitted in 1.3.1.1, item (2). Adding extensions to this structure, which may change the behavior of the application with respect to this standard when those fields in the structure are uninitialized, also requires that the extension be enabled as required by 1.3.1.1.

If the value described by *it_value* is nonzero, 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 is disarmed.

If the value described by *it_interval* is nonzero, it specifies an interval to 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 shall be disarmed after its next expiration—that is, a “one-shot” timer is specified.

Implementations may add extensions to these structures as permitted in 1.3.1.1, item (2). Adding extensions to this structure, which might change the behavior of the application with respect to this standard when those fields in the structures are uninitialized, also requires that the extensions be enabled as required by 1.3.1.1.

14.1.2 Timer Event Notification Control Block

For implementations that support the Realtime Signals Extension option, per-process timers may be created that notify the process of timer expirations by queuing a realtime extended signal. The *sigevent* structure, defined in `<signal.h>`, is used in creating such a timer. The *sigevent* structure contains the signal number and an application-specific data value to be used when notifying the calling process of timer expiration events.

14.1.3 Type Definitions

The following types are defined by the implementation in `<sys/types.h>`.

Defined Type	**Description**
clockid_t	Used for clock ID type in the clock and timer functions
timer_t	Used for timer ID returned by *timer_create*().

14.1.4 Manifest Constants

The following constants are defined in `<time.h>`:

CLOCK_REALTIME

The identifier for the systemwide realtime clock.

TIMER_ABSTIME

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

The maximum allowable resolution for the CLOCK_REALTIME clock and all timers based on this clock, including the *nanosleep*() function, is represented by {_POSIX_CLOCKRES_MIN} and is defined as 20 ms (1/50 of a second). Implementations may support smaller values of resolution for the CLOCK_REALTIME clock to provide finer granularity time bases. The actual resolution supported by an implementation for a specific clock is obtained using functions defined in this chapter. If the actual resolution supported for the *nanosleep*() function of timers based on this clock differs from the resolution supported for the clock, the implementation shall document this difference.

The minimum allowable maximum value for the CLOCK_REALTIME clock and absolute timers based on it is the same as that defined by the C Standard {2} for the *time_t* type. If the maximum value supported by the *nanosleep*() function or timers based on this clock is different than the maximum value supported by the clock, the implementation shall document this difference.

14.2 Clock and Timer Functions

14.2.1 Clocks

Functions: *clock_settime*(), *clock_gettime*(), *clock_getres*()

14.2.1.1 Synopsis

```
#include <time.h>
int clock_settime(clockid_t clock_id, const struct timespec *tp);
int clock_gettime(clockid_t clock_id, struct timespec *tp);
int clock_getres(clockid_t clock_id, struct timespec *res);
```

14.2.1.2 Description

If {_POSIX_TIMERS} is defined:

> The *clock_settime*() function shall set the specified clock, *clock_id*, to the value specified by *tp*. Time values that are between two consecutive nonnegative integer multiples of the resolution of the specified clock are truncated down to the smaller multiple of the resolution.
> The *clock_gettime*() function returns the current value *tp* for the specified clock, *clock_id*.
> The resolution of any clock can be obtained by calling *clock_getres*(). 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 is stored into 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*.
> A clock may be systemwide—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 defined in 14.1. 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.
> The effect of setting a clock via *clock_settime*() on armed per-process timers associated with that clock is implementation defined.
> The appropriate privilege to set a particular clock is implementation defined.

Otherwise:

> Either the implementation shall support the *clock_settime*(), *clock_gettime*(), and *clock_getres*() functions as described above or each of the *clock_settime*(), *clock_gettime*(), and *clock_getres*() functions shall fail.

14.2.1.3 Returns

A return value of 0 indicates that the call succeeded. A return value of –1 indicates that an error occurred, and *errno* is set to indicate the error.

14.2.1.4 Errors

If any of the following conditions occur, the *clock_settime*(), *clock_gettime*(), and *clock_getres*() functions shall return –1 and set *errno* to the corresponding value:

[EINVAL]	The *clock_id* argument does not specify a known clock.
[ENOSYS]	The functions *clock_settime*(), *clock_gettime*(), and *clock_getres*() are not supported by this implementation.

If any of the following conditions occur, the *clock_settime*() function shall return –1 and set *errno* to the corresponding value:

[EINVAL]	The *tp* argument to *clock_settime*() is outside the range for the given clock id.
	The *tp* argument specified a nanosecond value less than zero or greater than or equal to 1000 million.

For each of the following conditions, if the condition is detected, the *clock_settime*() function shall return −1 and set *errno* to the corresponding value:

[EPERM]	The requesting process does not have the appropriate privilege to set the specified clock.

14.2.1.5 Cross-References

timer_gettime(), 14.2.4; *time*(), 4.5.1; *ctime*(), 8.1.

14.2.2 Create a Per-Process Timer

Function: *timer_create*()

14.2.2.1 Synopsis

```
#include <signal.hh>
#include <time.h>
int timer_create(clockid_t clock_id, struct sigevent *evp,
           timer_t *timerid);
```

14.2.2.2 Description

If {_POSIX_TIMERS} is defined:

The *timer_create*() function shall create a per-process timer using the specified clock, *clock_id*, as the timing base. The *timer_create*() function returns, in the location referenced by *timerid*, a timer ID of type *timer_t* used to identify the timer in timer requests (see 14.2.4). 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, shall determine the asynchronous notification that will occur as specified in 3.3.1.2 when the timer expires. If the *evp* argument is **NULL**, the effect shall be as if the *evp* argument pointed to a *sigevent* structure where the *sigev_notify* member had the value SIGEV_SIGNAL, the *sigev_signo* had a default signal number, and the *sigev_value* member had the value of the timer ID.

Each implementation shall define a set of clocks that can be used as timing bases for per-process timers. All implementations shall support a *clock_id* of CLOCK_REALTIME.

Per-process timers shall not be inherited by a child process across a *fork*() and shall be disarmed and deleted by an *exec*.

Otherwise:

Either the implementation shall support the *timer_create*() function as described above or the *timer_create*() function shall fail.

14.2.2.3 Returns

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 (see 14.2.4). 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.

14.2.2.4 Errors

If any of the following conditions occur, the *timer_create*() function shall return −1 and set *errno* to the corresponding value:

[EAGAIN]	The system lacks sufficient signal queuing resources to honor the request.
	The calling process has already created all of the timers it is allowed by this implementation.
[EINVAL]	The specified clock ID is not defined.
[ENOSYS]	The function *timer_create*() is not supported by this implementation.

14.2.2.5 Cross-References

`<time.h>`, 14.1; *timer_delete*(), 14.2.3; *clock_gettime*(), 14.2.1; *clock_settime*(), 14.2.1; *clock_getres*(), 14.2.1; *timer_gettime*(), 14.2.4; *timer_settime*(), 14.2.4.

14.2.3 Delete a Per-Process Timer

Function: *timer_delete*()

14.2.3.1 Synopsis

```
#include <time.h>
int timer_delete(timer_t timerid);
```

14.2.3.2 Description

If {_POSIX_TIMERS} is defined:

> 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.

Otherwise:

> Either the implementation shall support the *timer_delete*() function as described above or the *timer_delete*() function shall fail.

14.2.3.3 Returns

If successful, the function shall return a value of zero. Otherwise, the function shall return a value of –1 and set *errno* to indicate the error.

14.2.3.4 Errors

If any of the following conditions occur, the *timer_delete*() function shall return –1 and set *errno* to the corresponding value:

[EINVAL]	The timer ID specified by *timerid* is not a valid timer ID.
[ENOSYS]	The function *timer_delete*() is not supported by this implementation.

14.2.3.5 Cross-References

timer_create(), 14.2.2.

14.2.4 Per-Process Timers

Functions: *timer_settime*(), *timer_gettime*(), *timer_getoverrun*()

14.2.4.1 Synopsis

```
#include <time.h>
int timer_settime(timer_t timerid, int flags,
            const struct itimerspec *value, struct itimerspec *ovalue);
int timer_gettime(timer_t timerid, struct itimerspec *value);
int timer_getoverrun(timer_t timerid);
```

14.2.4.2 Description

If {_POSIX_TIMERS} is defined:

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 nonzero. 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 on pending expiration notifications is unspecified.

If the flag TIMER_ABSTIME is not set in the argument *flags*, *timer_settime*() behaves 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 (see 14.1.1) from when the call is made. If the flag TIMER_ABSTIME is set in the argument *flags*, *timer_settime*() behaves 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 is set to the value specified by the *it_interval* member of *value*. When a timer is armed with a nonzero *it_interval*, a periodic (or repetitive) timer is specified.

Time values that are between two consecutive nonnegative 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 function *timer_settime*() 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. The members of *ovalue* are subject to the resolution of the timer, and they are the same values that would be returned by a *timer_gettime*() call at that point in time.

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*().

Only a single signal shall be queued to the process for a given timer at any point in time. When a timer for which a signal is still pending expires, no signal shall be queued, and a timer overrun shall occur. When a timer expiration signal is delivered to or accepted by a process, if the implementation supports the Realtime Signals Extension, the *timer_getoverrun*() function shall return the timer expiration overrun count for the specified timer. The overrun count returned shall contain 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*() applies to the most recent expiration signal delivery or acceptance for the timer. If no expiration signal has been delivered or accepted for the timer, or if the Realtime Signals Extension is not supported, the meaning of the overrun count returned is undefined.

Otherwise:

Either the implementation shall support the *timer_settime*(), *timer_gettime*(), and *timer_getoverrun*() functions as described above or each of the *timer_settime*(), *timer_gettime*(), and *timer_getoverrun*() functions shall fail.

14.2.4.3 Returns

If the *timer_settime*() or *timer_gettime*() functions succeed, a value of 0 shall be returned. If an error occurs for either of these functions, the value –1 shall be returned, and errno shall be set to indicate the error. If the *timer_getoverrun*() function succeeds, it shall return the timer expiration overrun count as explained in 14.2.4.2.

14.2.4.4 Errors

If any of the following conditions occur, the *timer_settime*(), *timer_gettime*(), and *timer_getoverrun*() functions shall return –1 and set *errno* to the corresponding value:

[EINVAL]	The *timerid* argument does not correspond to an id returned by *timer_create*() but not yet deleted by *timer_delete*().
[ENOSYS]	The functions *timer_settime*(), *timer_gettime*(), and *timer_getoverrun*() are not supported by this implementation.

If any of the following conditions occur, the *timer_settime*() function shall return –1 and set *errno* to the corresponding value:

[EINVAL]	A *value* structure specified a nanosecond value less than zero or greater than or equal to 1000 million.

14.2.4.5 Cross-References

clock_gettime(), 14.2.1; *timer_create*(), 14.2.2.

14.2.5 High Resolution Sleep

Function: *nanosleep*()

14.2.5.1 Synopsis

```
#include <time.h>
int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);
```

14.2.5.2 Description

If {_POSIX_TIMERS} is defined:

The *nanosleep*() function shall cause the current thread to be suspended from execution until either the time interval specified by the *rqtp* argument has elapsed, a signal is delivered to the calling thread and the action of the signal is to invoke a signal-catching function, or the process is terminated. The suspension time may be longer than requested because the argument value is rounded up to an integer multiple of the sleep resolution or because of the scheduling of other activity by the system. But, except for the case of being interrupted by

a signal, the suspension time shall not be less than the time specified by *rqtp*, as measured by the system clock, CLOCK_REALTIME.
The use of the *nanosleep*() function shall have no effect on the action or blockage of any signal.

Otherwise:

Either the implementation shall support the *nanosleep*() function as described above or the *nanosleep*() function shall fail.

14.2.5.3 Returns

If the *nanosleep*() function returns because the requested time has elapsed, its return value shall be zero.

If the *nanosleep*() function returns because it has been interrupted by a signal, the function shall return a value of –1 and set *errno* to indicate the interruption. If the *rmtp* argument is non-**NULL**, the *timespec* structure referenced by it 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.

If *nanosleep*() fails, it shall return a value of –1 and set *errno* to indicate the error.

14.2.5.4 Errors

If any of the following conditions occur, the *nanosleep*() function shall return –1 and set *errno* to the corresponding value:

[EINTR]	The *nanosleep*() function was interrupted by a signal.
[EINVAL]	The *rqtp* argument specified a nanosecond value less than zero or greater than or equal to 1000 million.
[ENOSYS]	The *nanosleep*() function is not supported by this implementation.

14.2.5.5 Cross-References

sleep(), 3.4.3.

15. Message Passing

15.1 Data Definitions for Message Queues

Inclusion of the `<mqueue.h>` header may make visible the symbols allowed by this part of ISO/IEC 9945 to be in the headers `<sys/types.h>`, `<fcntl.h>`, `<time.h>`, and `<signal.h>`.

15.1.1 Data Structures

The header `<mqueue.h>` shall define the following implementation-defined types:

Type	Description
mqd_t	Used for message queue descriptors.

The type *mqd_t* shall not be an array type. The message queue descriptor may be implemented using a file descriptor. In that case, applications shall be able to open up to at least a total of {OPEN_MAX} file and message queues; see 5.3.1.

The header `<mqueue.h>` defines the following implementation-defined structures:

Type	Description
struct sigevent	As specified in 3.3.1.

A message queue status structure *mq_attr* is used in getting and setting the attributes of a message queue. Attributes are initially set when the message queue is created. This structure is defined in `<mqueue.h>` and has at least the following members:

Member Type	Member Name	Description
long	*mq_flags*	Message queue flags.
long	*mq_maxmsg*	Maximum number of messages.
long	*mq_msgsize*	Maximum message size.
long	*mq_curmsgs*	Number of messages currently queued.

Implementations may add extensions as permitted in 1.3.1.1, item (2). Adding extensions to this structure, which may change the behavior of the application with respect to this standard when those fields in the structure are uninitialized, also requires that the extension be enabled as required by 1.3.1.1.

A brief description of each of the above members is given below; further details are given in the message queue functions described later in this section. The following members represent attributes that can be set and queried.

mq_flags Specifies actions and state for the message queue operations. The following flags shall be defined:

O_NONBLOCK

If set, then *mq_send*() or *mq_receive*() operations associated with this message queue shall not block.

The existence of other flags is unspecified.

The following members represent attributes that can be queried, but they can only be set at message queue creation.

mq_maxmsg

Specifies the number of messages that can be held in the message queue without causing *mq_send*() to fail or wait due to lack of resources.

mq_msgsize

Specifies the maximum size of each message in the message queue.

The following members represent the current status of dynamic attributes of the message queue. These can be queried, but they cannot be explicitly set.

mq_curmsgs	Indicates the number of messages currently on the queue.

15.2 Message Passing Functions

15.2.1 Open a Message Queue

Function: *mq_open*()

15.2.1.1 Synopsis

```
#include <mqueue.h>
mqd_t mq_open(const char *name, int oflag, ...);
```

15.2.1.2 Description

If {_POSIX_MESSAGE_PASSING} is defined:

The *mq_open*() function establishes the connection between a process and a message queue with a message queue descriptor. It creates an open message queue description that refers to the message queue, and it creates a message queue descriptor that refers to that open message queue description. The *name* argument points to a string naming a message queue. It is unspecified whether the name appears in the file system and is visible to other functions that take pathnames as arguments. The *name* argument shall conform to the construction rules for a pathname. If *name* begins with the slash character, then processes calling *mq_open*() with the same value of name shall refer to the same message queue object, as long as that name has not been removed. If *name* does not begin with the slash character, the effect is implementation defined. The interpretation of slash characters other than the leading slash character in *name* is implementation defined. If the *name* argument is not the name of an existing message queue and creation is not requested, *mq_open*() shall fail and return an error.

The *oflag* argument requests the desired receive and/or send access to the message queue. The requested access permission to receive messages or send messages is granted if the calling process would be granted read or write access, respectively, to an equivalently protected file. Read and write access to the file is determined as described in 2.3.

The value of *oflag* is the bitwise inclusive OR of values from the following list. Applications shall specify exactly one of the first three values (access modes) below in the value of *oflag*:

O_RDONLY	Open the message queue for receiving messages. The process can use the returned message queue descriptor with *mq_receive*(), but not *mq_send*(). A message queue may be open multiple times in the same or different processes for receiving messages.
O_WRONLY	Open the queue for sending messages. The process can use the returned message queue descriptor with *mq_send*() but not *mq_receive*(). A message queue may be open multiple times in the same or different processes for sending messages.
O_RDWR	Open the queue for both receiving and sending messages. The process can use any of the functions allowed for O_RDONLY and O_WRONLY. A message queue may be open multiple times in the same or different processes for sending messages.

Any combination of the remaining flags may be specified in the value of *oflag*:

O_CREAT	This option is used to create a message queue, and it requires two additional arguments: *mode*, which is of type *mode_t*, and *attr*, which is a pointer to a *mq_attr* structure. If the pathname, *name*, has already been used to create a message queue that still exists, then this flag has no effect, except as noted under O_EXCL. Otherwise, a message queue is created

without any messages in it. The user ID of the message queue shall be set to the effective user ID of the process, and the group ID of the message queue shall be set to the effective group ID of the process. The "file permission bits" shall be set to the value of *mode*. When bits in *mode* other than file permission bits are set, the effect is implementation defined. If *attr* is **NULL**, the message queue is created with implementation-defined default message queue attributes. If *attr* is non-**NULL** and the calling process has the appropriate privilege on *name*, the message queue *mq_maxmsg* and *mq_msgsize* attributes are set to the values of the corresponding members in the *mq_attr* structure referred to by *attr*. If *attr* is non-**NULL**, but the calling process does not have the appropriate privilege on *name*, the *mq_open*() function shall fail and return an error without creating the message queue.

O_EXCL If O_EXCL and O_CREAT are set, *mq_open*() shall fail if the message queue *name* exists. The check for the existence of the message queue and the creation of the message queue if it does not exist shall be atomic with respect to other processes executing *mq_open*() naming the same *name* with O_EXCL and O_CREAT set. If O_EXCL is set and O_CREAT is not set, the result is undefined.

O_NONBLOCK The setting of this flag is associated with the open message queue description and determines whether a *mq_send*() or *mq_receive*() shall wait for resources or messages that are not currently available, or fail with *errno* set to [EAGAIN]. See *mq_send*() and *mq_receive*() for details.

The *mq_open*() function shall not add or remove messages from the queue.

Otherwise:

Either the implementation shall support the *mq_open*() function as described above or the *mq_open*() function shall fail.

15.2.1.3 Returns

Upon successful completion, the function shall return a message queue descriptor. Otherwise, the function shall return (*mqd_t*) −1 and set *errno* to indicate the error.

15.2.1.4 Errors

If any of the following conditions occur, the *mq_open*() function shall return (*mqd_t*) −1 and set *errno* to the corresponding value:

[EACCES] The message queue exists and the permissions specified by *oflag* are denied, or the message queue does not exist and permission to create the message queue is denied.

[EEXIST] O_CREAT and O_EXCL are set and the named message queue already exists.

[EINTR] The *mq_open*() operation was interrupted by a signal.

[EINVAL] The *mq_open*() operation is not supported for the given name. The implementation shall document under what circumstances this error may be returned.

O_CREAT was specified in *oflag*, the value of *attr* is not **NULL**, and either *mq_maxmsg* or *mq_msgsize* was less than or equal to zero.

[EMFILE] Too many message queue descriptors or file descriptors are currently in use by this process.

[ENAMETOOLONG]

The length of the *name* string exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.

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

[ENOENT]	O_CREAT is not set and the named message queue does not exist.
[ENOSPC]	There is insufficient space for the creation of the new message queue.
[ENOSYS]	The function *mq_open*() is not supported by this implementation.

15.2.1.5 Cross-References

mq_close(), 15.2.2; *mq_receive*(), 15.2.5; *mq_send*(), 15.2.4; *mq_setattr*(), 15.2.7; *mq_getattr*(), 15.2.8; *mq_unlink*(), 15.2.3.

15.2.2 Close a Message Queue

Function: *mq_close*()

15.2.2.1 Synopsis

```
#include <mqueue.h>
int mq_close(mqd_t mqdes);
```

15.2.2.2 Description

If {_POSIX_MESSAGE_PASSING} is defined:

> The *mq_close*() function shall remove the association between the message queue descriptor, *mqdes*, and its message queue. The results of using this message queue descriptor after successful return from this *mq_close*(), and until the return of this message queue descriptor from a subsequent *mq_open*(), are undefined.
> If the process has successfully attached a notification request to the message queue via this *mqdes*, this attachment shall be removed, and the message queue is available for another process to attach for notification.

Otherwise:

> Either the implementation shall support the *mq_close*() function as described above or the *mq_close*() function shall fail.

15.2.2.3 Returns

Upon successful completion, the *mq_close*() function shall return a value of zero; otherwise, the function shall return a value of –1 and set *errno* to indicate the error.

15.2.2.4 Errors

If any of the following conditions occur, the *mq_close*() function shall return –1 and set *errno* to the corresponding value:

[EBADF]	The *mqdes* argument is not a valid message queue descriptor.
[ENOSYS]	The function *mq_close*() is not supported by this implementation.

15.2.2.5 Cross-References

mq_open(), 15.2.1; *mq_unlink*(), 15.2.3.

15.2.3 Remove a Message Queue

Function: *mq_unlink*()

15.2.3.1 Synopsis

```
#include <mqueue.h>
int mq_unlink(const char *name);
```

15.2.3.2 Description

If {_POSIX_MESSAGE_PASSING} is defined:

> 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 re-create 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.

Otherwise:

> Either the implementation shall support the *mq_unlink*() function as described above or the *mq_unlink*() function shall fail.

15.2.3.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, the named message queue shall not be changed by this function call, and the function shall return a value of –1 and set *errno* to indicate the error.

15.2.3.4 Errors

If any of the following conditions occur, the *mq_unlink*() function shall return –1 and set *errno* to the corresponding value:

[EACCES]	Permission is denied to unlink the named message queue.
[ENAMETOOLONG]	The length of the *name* string exceeds {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.
[ENOENT]	The named message queue does not exist.
[ENOSYS]	The function *mq_unlink*() is not supported by this implementation.

15.2.3.5 Cross-References

mq_close(), 15.2.2; *mq_open*(), 15.2.1.

15.2.4 Send a Message to a Message Queue

Function: *mq_send*()

15.2.4.1 Synopsis

```
#include <mqueue.h>
```

```
int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
            unsigned int msg_prio);
```

15.2.4.2 Description

If {_POSIX_MESSAGE_PASSING} is defined:

> The *mq_send*() function adds 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* is 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 Process 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 is not queued and *mq_send*() returns an error.

Otherwise:

> Either the implementation shall support the *mq_send*() function as described above or the *mq_send*() function shall fail.

15.2.4.3 Returns

Upon successful completion, the *mq_send*() function shall return a value of zero. Otherwise, no message shall be enqueued, the function shall return −1, and *errno* shall be set to indicate the error.

15.2.4.4 Errors

If any of the following conditions occur, the *mq_send*() function shall return −1 and set *errno* to the corresponding value:

[EAGAIN]	The O_NONBLOCK flag is set in the message queue description associated with *mqdes*, and the specified message queue is full.
[EBADF]	The *mqdes* argument is not a valid message queue descriptor open for writing.
[EINTR]	A signal interrupted the call to *mq_send*().
[EINVAL]	The value of *msg_prio* was outside the valid range.
[EMSGSIZE]	The specified message length, *msg_len*, exceeds the message size attribute of the message queue.
[ENOSYS]	The function *mq_send*() is not supported by this implementation.

15.2.4.5 Cross-References

mq_receive(), 15.2.5; *mq_setattr*(), 15.2.7.

15.2.5 Receive a Message From a Message Queue

Function: *mq_receive*()

15.2.5.1 Synopsis

```
#include <mqueue.h>
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
            unsigned int *msg_prio);
```

15.2.5.2 Description

If {_POSIX_MESSAGE_PASSING} is defined:

The *mq_receive*() function is used to 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 is removed from the queue and copied to the buffer pointed to by the *msg_ptr* argument.
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 Process 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 is removed from the queue, and *mq_receive*() returns an error.

Otherwise:

Either the implementation shall support the *mq_receive*() function as described above or the *mq_receive*() function shall fail.

15.2.5.3 Returns

Upon successful completion, *mq_receive*() shall return the length of the selected message in bytes and the message shall have been removed from the queue. Otherwise, no message shall be removed from the queue, the function shall return a value of –1, and set *errno* to indicate the error.

15.2.5.4 Errors

If any of the following conditions occur, the *mq_receive*() function shall return –1 and set *errno* to the corresponding value:

[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*() operation was interrupted by a signal.
[ENOSYS]	The *mq_receive*() function is not supported by this implementation.

For each of the following conditions, if the condition is detected, the *mq_receive*() function shall return −1 and set *errno* to the corresponding value:

[EBADMSG] The implementation has detected a data corruption problem with the message.

15.2.5.5 Cross-References

mq_send(), 15.2.4.

15.2.6 Notify Process That a Message is Available on a Queue

Function: *mq_notify*()

15.2.6.1 Synopsis

```
#include <mqueue.h>
int mq_notify(mqd_t mqdes, const struct sigevent *notification);
```

15.2.6.2 Description

If {_POSIX_MESSAGE_PASSING} and {_POSIX_REALTIME_SIGNALS} are defined:

> If the argument *notification* is not **NULL**, this function registers the calling process to be notified of message arrival at an empty message queue associated with the specified message queue descriptor, *mqdes*. The notification specified by the *notification* argument shall be sent to the process when the message queue transitions from empty to nonempty. At any time, only one process may be registered for notification by a message queue. If the calling process or any other process has already registered for notification of message arrival at the specified message queue, subsequent attempts to register for that message queue shall fail.
> If *notification* is **NULL** and the process is currently registered for notification by the specified message queue, the existing registration shall be removed.
> When the notification is sent to the registered process, its registration shall be removed. The message queue shall then be available for registration.
> If a process has registered for notification of message arrival at a message queue and some thread is blocked in *mq_receive*() waiting to receive a message when a message arrives at the queue, the arriving message shall satisfy the appropriate *mq_receive*() (see 15.2.5). The resulting behavior is as if the message queue remains empty, and no notification shall be sent.

Otherwise:

> Either the implementation shall support the *mq_notify*() function as described above or the *mq_notify*() function shall fail.

15.2.6.3 Returns

Upon successful completion, the *mq_notify*() function shall return a value of zero; otherwise, the function shall return a value of –1 and set *errno* to indicate the error.

15.2.6.4 Errors

If any of the following conditions occur, the *mq_notify*() function shall return –1 and set *errno* to the corresponding value:

[EBADF] The *mqdes* argument is not a valid message queue descriptor.

[EBUSY]	A process is already registered for notification by the message queue.
[ENOSYS]	The function *mq_notify*() is not supported by this implementation.

15.2.6.5 Cross-References

mq_open(), 15.2.1; *mq_send*(), 15.2.4.

15.2.7 Set Message Queue Attributes

Function: *mq_setattr*()

15.2.7.1 Synopsis

```
#include <mqueue.h>
int mq_setattr(mqd_t mqdes, const struct mq_attr *mqstat,
            struct mq_attr *omqstat);
```

15.2.7.2 Description

If {_POSIX MESSAGE_PASSING} is defined:

The *mq_setattr*() function is used to 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 are 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 are ignored by *mq_setattr*().
If *omqstat* is non-**NULL**, the function *mq_setattr*() shall store, in the location referenced by *omqstat,* the previous message queue attributes and the current queue status. These values are the same as would be returned by a call to *mq_getattr*() at that point.

Otherwise:

Either the implementation shall support the *mq_setattr*() function as described above or the *mq_setattr*() function shall fail.

15.2.7.3 Returns

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.

15.2.7.4 Errors

If any of the following conditions occur, the *mq_setattr*() function shall return –1 and set *errno* to the corresponding value:

[EBADF]	The *mqdes* argument is not a valid message queue descriptor.
[ENOSYS]	The function *mq_setattr*() is not supported by this implementation.

15.2.7.5 Cross-References

mq_open(), 15.2.1; *mq_send*(), 15.2.4.

15.2.8 Get Message Queue Attributes

Function: *mq_getattr*()

15.2.8.1 Synopsis

```
#include <mqueue.h>
int mq_getattr(mqd_t mqdes, struct mq_attr *mqstat);
```

15.2.8.2 Description

If {_POSIX_MESSAGE_PASSING} is defined:

> The *mqdes* argument specifies a message queue descriptor. The *mq_getattr*() function is used to get status information and attributes of the message queue and the open message queue description associated with the message queue descriptor. The results are returned in the *mq_attr* structure referenced by the *mqstat* argument.
> Upon return, the following members shall have the values associated with the open message queue description as set when the message queue was opened and as modified by subsequent *mq_setattr*() calls.
>
> *mq_flags*
>
> The following attributes of the message queue shall be returned as set at message queue creation.
>
> *mq_maxmsg*
> *mq_msgsize*
>
> Upon return, the following members within the *mq_attr* structure referenced by the *mqstat* argument shall be set according to the current state of the message queue.

mq_curmsgs	The number of messages currently on the queue.

Otherwise:

> Either the implementation shall support the *mq_getattr*() function as described above or the *mq_getattr*() function shall fail.

15.2.8.3 Returns

Upon successful completion, the *mq_getattr*() function shall return zero. Otherwise, the function shall return −1 and set *errno* to indicate the error.

15.2.8.4 Errors

If any of the following conditions occur, the *mq_getattr*() function shall return −1 and set *errno* to the corresponding value:

[EBADF]	The *mqdes* argument is not a valid message queue descriptor.
[ENOSYS]	The function *mq_getattr*() is not supported by this implementation.

15.2.8.5 Cross-References

`<mqueue.h>`, 15.1.1; *mq_open*(), 15.2.1; *mq_send*(), 15.2.4; *mq_setattr*() 15.2.7.

16. Thread Management

16.1 Threads

This section describes the facilities relating to multiple threads of control available through this standard.

A thread is a single flow of control within a process. This section defines a set of operations that allow for the creation and management of multiple threads within a single process.

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 by this part of ISO/IEC 9945 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*().

16.2 Thread Functions

16.2.1 Thread Creation Attributes

Functions: *pthread_attr_init*(), *pthread_attr_destroy*(), *pthread_attr_setstacksize*(), *pthread_attr_getstacksize*(), *pthread_attr_setstackaddr*(), *pthread_attr_getstackaddr*(), *pthread_attr_setdetachstate*(), *pthread_attr_getdetachstate*()

16.2.1.1 Synopsis

```
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
int pthread_attr_getstacksize(const pthread_attr_t *attr,
            size_t *stacksize);
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
int pthread_attr_getstackaddr(const pthread_attr_t *attr,
            void **stackaddr);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
int pthread_attr_getdetachstate(const pthread_attr_t *attr,
            int *detachstate);
```

16.2.1.2 Description

If {_POSIX_THREADS} is defined:

> The function *pthread_attr_init*() initializes a thread attributes object *attr* with the default value for all of the individual attributes used by a given implementation.
> Each implementation shall document the individual attributes it uses and their default values unless these values are defined by this standard.
> 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*().

The *pthread_attr_destroy*() function is used to destroy a thread attributes object. An implementation may cause *pthread_attr_destroy*() to set *attr* to an implementation-specific invalid value. The behavior of using the attribute after it has been destroyed is undefined.

If the symbol {_POSIX_THREAD_ATTR_STACKSIZE} is defined, then the implementation shall support a stacksize attribute for threads that defines the minimum stack size (in bytes). The functions *pthread_attr_setstacksize*() and *pthread_attr_getstacksize*() set and get the thread creation stacksize attribute in the *attr* object.

If the symbol {_POSIX_THREAD_ATTR_STACKADDR} is defined, then the implementation shall support a stackaddr attribute that specifies the location of storage to be used for the stack of the created thread. The size of the storage shall be at least {PTHREAD_STACK_MIN}. The functions *pthread_attr_setstackaddr*() and *pthread_attr_getstackaddr*() set and get the thread creation stackaddr attribute in the *attr* object.

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*() functions is an error.

The *pthread_attr_setdetachstate*() and *pthread_attr_getdetachstate*() functions set and get the `detachstate` attribute in the *attr* object. The location referenced by the *detachstate* argument shall be set 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 is PTHREAD_CREATE_JOINABLE.

Otherwise:

Either the implementation shall support the *pthread_attr_init*(), *pthread_attr_destroy*(), *pthread_attr_setstacksize*(), *pthread_attr_getstacksize*(), *pthread_attr_setstackaddr*(), *pthread_attr_-getstackaddr*(), *pthread_attr_setdetachstate*(), and *pthread_attr_getdetachstate*() functions as described above or the *pthread_attr_init*(), *pthread_attr_destroy*(), *pthread_attr_setstacksize*(), *pthread__attr_-getstacksize*(), *pthread_attr_setstackaddr*(), *pthread_attr_getstackaddr*(), *pthread_attr_setdetachstate*(), and *pthread_attr_getdetachstate*() functions shall not be provided.

16.2.1.3 Returns

Upon successful completion, *pthread_attr_init*(), *pthread_attr_destroy*(), *pthread_attr_setstacksize*(), *pthread_attr_-getstacksize*(), *pthread_attr_setstackaddr*(), *pthread_attr_getstackaddr*(), *pthread_attr_setdetachstate*(), and *pthread_attr_getdetachstate*() 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. The *pthread_attr_getstackaddr*() function stores the stackaddr attribute value in *stackaddr* if successful. The *pthread_attr_getdetachstate*() function stores the value of the `detachstate` attribute in *detachstate* if successful.

16.2.1.4 Errors

If any of the following conditions occur, the *pthread_attr_init*() function shall return the corresponding error number:

[ENOMEM]	Insufficient memory exists to initialize the thread attributes object.

If any of the following conditions occur, the *pthread_attr_getstacksize*() and *pthread_attr_setstacksize*() functions shall return the corresponding error number:

[ENOSYS]	The option {_POSIX_THREAD_ATTR_STACKSIZE} is not defined and the stacksize attribute for threads is not supported.

If any of the following conditions occur, the *pthread_attr_setstacksize*() function shall return the corresponding error number:

[EINVAL] The value of *stacksize* is less than {PTHREAD_STACK_MIN} or exceeds a system-imposed limit.

If any of the following conditions occur, the *pthread_attr_getstackaddr*() and *pthread_attr_setstackaddr*() functions shall return the corresponding error number:

[ENOSYS] The option {_POSIX_THREAD_ATTR_STACKADDR} is not defined, and the stackaddr attribute for threads is not supported.

If any of the following conditions occur, the *pthread_attr_setdetachstate*() function shall return the corresponding error number:

[EINVAL] The value of *detachstate* was not valid.

16.2.1.5 Cross-References

pthread_create(), 16.2.2.

16.2.2 Thread Creation

Function: *pthread_create*()

16.2.2.1 Synopsis

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
            void *(*start_routine) (void *), void *arg);
```

16.2.2.2 Description

If {_POSIX_THREADS} is defined:

> The *pthread_create*() function is used to create a new thread, with attributes specified by *attr*, within a process. If *attr* is **NULL**, the default attributes are used. If the attributes specified by *attr* are modified later, the attributes of the thread are not affected. Upon successful completion, *pthread_create*() shall store the ID of the created thread in the location referenced by *thread*.
> The thread is created executing *start_routine* with *arg* as its sole argument. If the *start_routine* returns, the effect shall be as if there was an implicit call to *pthread_exit*() using the return value of *start_routine* as the exit status. Note that the thread in which *main*() was originally invoked differs from this. When this thread returns from *main*(), the effect shall be as if there was an implicit call to *exit*() using the return value of *main*() as the exit status.
> The signal state of the new thread shall be initialized as follows:
> 1) The signal mask shall be inherited from the creating thread.
> 2) The set of signals pending for the new thread shall be empty.
>
> If *pthread_create*() fails, no new thread is created, and the contents of the location referenced by *thread* are undefined.

Otherwise:

> Either the implementation shall support the *pthread_create*() function as described above or the *pthread_create*() function shall not be provided.

16.2.2.3 Returns

If successful, the *pthread_create*() function shall return zero. Otherwise, an error number shall be returned to indicate the error.

16.2.2.4 Errors

If any of the following conditions occur, the *pthread_create*() function shall return the corresponding error number:

[EAGAIN]	The system lacked the necessary resources to create another thread, or the system-imposed limit on the total number of threads in a process {PTHREAD_THREADS_MAX} would be exceeded.
[EINVAL]	The value specified by *attr* is invalid.

16.2.2.5 Cross-References

pthread_exit(), 16.2.5; *pthread_join*(), 16.2.3; *fork*(), 3.1.1.

16.2.3 Wait for Thread Termination

Function: *pthread_join*()

16.2.3.1 Synopsis

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **value_ptr);
```

16.2.3.2 Description

If {_POSIX_THREADS} is defined:

> The *pthread_join*() function suspends 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}.

Otherwise:

> Either the implementation shall support the *pthread_join*() function as described above or the *pthread_join*() function shall not be provided.

16.2.3.3 Returns

If successful, the *pthread_join*() function shall return zero. Otherwise, an error number shall be returned to indicate the error.

16.2.3.4 Errors

If any of the following conditions occur, the *pthread_join*() function shall return the corresponding error number:

[EINVAL]	The implementation has detected that the value specified by *thread* does not refer to a thread that can be joined.
[ESRCH]	No thread could be found corresponding to that specified by the given thread ID.

For each of the following conditions, if the condition is detected, the *pthread_join*() function shall return the corresponding error number:

[EDEADLK]	A deadlock was detected, or the value of *thread* specifies the calling thread.

16.2.3.5 Cross-References

pthread_create(), 16.2.2; *wait*(), 3.2.1.

16.2.4 Detaching a Thread

Function: *pthread_detach*()

16.2.4.1 Synopsis

```
#include <pthread.h>
int pthread_detach(pthread_t thread);
```

16.2.4.2 Description

If {_POSIX_THREADS} is defined:

> The *pthread_detach*() function is used to indicate to the implementation that storage for the thread *thread* can be reclaimed when that thread terminates. If *thread* has not terminated, *pthread_detach*() shall not cause it to terminate. The effect of multiple *pthread_detach*() calls on the same target thread is unspecified.

Otherwise:

> Either the implementation shall support the *pthread_detach*() function as described above or the *pthread_detach*() function shall not be provided.

16.2.4.3 Returns

If the call succeeds, *pthread_detach*() returns 0. Otherwise, an error number shall be returned to indicate the error.

16.2.4.4 Errors

If any of the following conditions occur, the *pthread_detach*() function shall return the corresponding error number:

[EINVAL]	The implementation has detected that the value specified by *thread* does not refer to a thread that can be joined.
[ESRCH]	No thread could be found corresponding to that specified by the given thread ID.

16.2.4.5 Cross-References

pthread_join(), 16.2.3.

16.2.5 Thread Termination

Function: *pthread_exit*()

16.2.5.1 Synopsis

```
#include <pthread.h>
void pthread_exit(void *value_ptr);
```

16.2.5.2 Description

If {_POSIX_THREADS} is defined:

> The *pthread_exit*() function terminates the calling thread and makes 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 return value of the function serves as the exit status of the thread.
>
> 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 the time of thread termination.

Otherwise:

> Either the implementation shall support the *pthread_exit*() function as described above or the *pthread_exit*() function shall not be provided.

16.2.5.3 Returns

The *pthread_exit*() function cannot return to its caller.

16.2.5.4 Errors

None.

16.2.5.5 Cross-References

pthread_create(), 16.2.2; *pthread_join*(), 16.2.3; *exit*(), 8.1; *_exit*(), 3.2.2.

16.2.6 Get Thread ID

Function: *pthread_self*()

16.2.6.1 Synopsis

```
#include <pthread.h>
pthread_t pthread_self(void);
```

16.2.6.2 Description

If {_POSIX_THREADS} is defined:

> The *pthread_self*() function returns the thread ID of the calling thread.

16.2.6.3 Returns

See 16.2.6.2.

16.2.6.4 Errors

None.

16.2.6.5 Cross-References

pthread_create(), 16.2.2; *pthread_equal*(), 16.2.7.

16.2.7 Compare Thread IDs

Function: *pthread_equal*()

16.2.7.1 Synopsis

```
#include <pthread.h>
int pthread_equal(pthread_t t1, pthread_t t2);
```

16.2.7.2 Description

If {_POSIX_THREADS} is defined:

> This function compares the thread IDs *t1* and *t2*.

Otherwise:

> Either the implementation shall support the *pthread_equal*() function as described above or the *pthread_equal*() function shall not be provided.

16.2.7.3 Returns

The *pthread_equal*() function shall return a nonzero value if *t1* and *t2* are equal; otherwise, zero shall be returned.

If either *t1* or *t2* are not valid thread IDs, the behavior is undefined.

16.2.7.4 Errors

None.

16.2.7.5 Cross-References

pthread_create(), 16.2.2; *pthread_self*(), 16.2.6.

16.2.8 Dynamic Package Initialization

Function: *pthread_once*()

16.2.8.1 Synopsis

```
#include <pthread.h>
pthread_once_t once_control = PTHREAD_ONCE_INIT;
int pthread_once(pthread_once_t *once_control,
            void (*init_routine) (void));
```

16.2.8.2 Description

If {_POSIX_THREADS} is defined:

> The first call to *pthread_once*() by any thread in a process with a given *once_control* will call the *init_routine*() with no arguments. Subsequent calls of *pthread_once*() with the same *once_control* will not call the *init_routine*(). On return from *pthread_once*(), it is guaranteed that *init_routine*() has completed. The *once_control* parameter is used to determine whether the associated initialization routine has been called.
> The function *pthread_once*() 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 shall be defined by the header `<pthread.h>`.
> The behavior of *pthread_once*() is undefined if *once_control* has automatic storage duration or is not initialized by PTHREAD_ONCE_INIT.

Otherwise:

> Either the implementation shall support the *pthread_once*() function as described above or the *pthread_once*() function shall not be provided.

16.2.8.3 Returns

Upon successful completion, *pthread_once*() shall return zero. Otherwise, an error number shall be returned to indicate the error.

16.2.8.4 Errors

None specified.

17. Thread-Specific Data

This section describes the facilities available in this part of ISO/IEC 9945 relating to the association of threads and data.

17.1 Thread-Specific Data Functions

17.1.1 Thread-Specific Data Key Creation

Function: *pthread_key_create*()

17.1.1.1 Synopsis

```
#include <pthread.h>
int pthread_key_create(pthread_key_t *key, void (*destructor) (void *));
```

17.1.1.2 Description

If {_POSIX_THREADS} is defined:

> This function creates a thread-specific data key visible to all threads in the process. Key values provided by *pthread_key_create*() are opaque objects used to locate thread-specific data. Although the same key value may be used by different threads, the values bound to the key by *pthread_setspecific*() are maintained on a per-thread basis and persist for the life of the calling thread.
> Upon key creation, the value **NULL** shall be associated with the new key in all active threads. Upon thread creation, the value **NULL** shall be associated with all defined keys in the new thread.
> An optional destructor function may be associated with each key value. At thread exit, if a key value has a non-**NULL** destructor pointer and if the thread has a non-**NULL** value associated with that key, the function pointed to is called with the current associated value as its sole argument. The order of destructor calls is unspecified if more than one destructor exists for a thread when it exits.
> If, after all the destructors have been called for all non-**NULL** values with associated destructors, there are still some non-**NULL** values with associated destructors, then the process shall be repeated. If, after at least {PTHREAD_DESTRUCTOR_ITERATIONS} iterations of destructor calls for outstanding non-**NULL** values, there are still some non-**NULL** values with associated destructors, implementations may stop calling destructors, or they may continue calling destructors until no non-**NULL** values with associated destructors exist, even though this might result in an infinite loop.

Otherwise:

> Either the implementation shall support the *pthread_key_create*() function as described above or the *pthread_key_create*() function shall not be provided.

17.1.1.3 Returns

If successful, the *pthread_key_create*() function shall store the newly created key value at **key* and return zero. Otherwise, an error number shall be returned to indicate the error.

17.1.1.4 Errors

If any of the following conditions occur, the *pthread_key_create*() function shall return the corresponding error number:

[EAGAIN]	The system lacked the necessary resources to create another thread-specific data key, or the system-imposed limit on the total number of keys per process {PTHREAD_KEYS_MAX} has been exceeded.
[ENOMEM]	Insufficient memory exists to create the key.

17.1.1.5 Cross-References

pthread_getspecific(), 17.1.2; *pthread_setspecific*(), 17.1.2; *pthread_key_delete*(), 17.1.3.

17.1.2 Thread-Specific Data Management

Functions: *pthread_setspecific*(), *pthread_getspecific*()

17.1.2.1 Synopsis

```
#include <pthread.h>
int pthread_setspecific(pthread_key_t key, const void *value);
void *pthread_getspecific(pthread_key_t key);
```

17.1.2.2 Description

If {_POSIX_THREADS} is defined:

> The *pthread_setspecific*() function associates a thread-specific *value* with a *key* obtained via a previous call to *pthread_key_create*(). Different threads may bind different values to the same key. These values are typically pointers to blocks of dynamically allocated memory that have been reserved for use by the calling thread.
> The *pthread_getspecific*() function returns the value currently bound to the specified *key* on behalf of the calling thread.
> The effect of calling *pthread_setspecific*() or *pthread_getspecific*() with a *key* value not obtained from *pthread_key_create*() or after *key* has been deleted with *pthread_key_delete*() is undefined.
> Both *pthread_setspecific*() and *pthread_getspecific*() may be called from a thread-specific data destructor function. However, calling *pthread_setspecific*() from a destructor may result in lost storage or infinite loops.
> Both functions may be implemented as macros.

Otherwise:

> Either the implementation shall support the *pthread_setspecific*() and *pthread_getspecific*() functions as described above or the *pthread_setspecific*() and *pthread_getspecific*() functions shall not be provided.

17.1.2.3 Returns

The function *pthread_getspecific*() returns the thread-specific data value associated with the given *key*. If no thread-specific data value is associated with *key*, then the value **NULL** is returned.

If successful, the *pthread_setspecific*() function shall return zero. Otherwise, an error number shall be returned to indicate the error.

17.1.2.4 Errors

If any of the following conditions occur, the *pthread_setspecific*() function shall return the corresponding error number:

[ENOMEM]	Insufficient memory exists to associate the value with the key.

For each of the following conditions, if the condition is detected, the *pthread_setspecific*() function shall return the corresponding error number:

[EINVAL]	The key value is invalid.

No errors are returned from *pthread_getspecific*().

17.1.2.5 Cross-References

pthread_key_create(), 17.1.1.

17.1.3 Thread-Specific Data Key Deletion

Function: *pthread_key_delete*()

17.1.3.1 Synopsis

```
#include <pthread.h>
int pthread_key_delete(pthread_key_t key);
```

17.1.3.2 Description

If {_POSIX_THREADS} is defined:

> This function deletes 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.

Otherwise:

> Either the implementation shall support the *pthread_key_delete*() function as described above or the *pthread_key_delete*() function shall not be provided.

17.1.3.3 Returns

If successful, the *pthread_key_delete*() function shall return zero. Otherwise, an error number shall be returned to indicate the error.

17.1.3.4 Errors

For each of the following conditions, if the condition is detected, the *pthread_key_delete*() function shall return the corresponding error number:

[EINVAL]	The *key* value is invalid.

17.1.3.5 Cross-References

pthread_key_create(), 17.1.1.

18. Thread Cancellation

This section defines a set of operations that allow for the cancellation of threads.

The C language binding to thread cancellation is presented first, followed by a presentation of the language-independent thread cancellation functionality required of all language bindings.

18.1 Thread Cancellation Overview

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 being canceled) is allowed to hold cancellation requests pending in a number of ways and to perform application-specific cleanup processing when acting on the notice of cancellation.

Cancellation is controlled by the cancellation control interfaces. 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 *deferred* cancelability set, which is specified as the default. Applications also need to follow carefully static lexical scoping rules in their execution behavior. For instance, use of *setjmp*(), return, goto, etc. to leave user-defined cancellation scopes without doing the necessary scope pop operation will result in undefined behavior.

Use of asynchronous cancelability while holding resources that 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.

18.1.1 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:

> *Cancelability Enable*: When cancelability is PTHREAD_CANCEL_DISABLE, cancellation requests against the target thread are held pending. By default, cancelability is set to PTHREAD_CANCEL_ENABLE.
> *Cancelability Type*: When cancelability is enabled and the cancelability type is PTHREAD_CANCEL_ASYNCHRONOUS, new or pending cancellation requests may be acted upon at any time. When cancelability is enabled and the cancelability type is PTHREAD_CANCEL_DEFERRED, 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 will be in effect. The cancelability type is PTHREAD_CANCEL_DEFERRED in all newly created threads, including the thread in which *main*() was first invoked.

18.1.2 Cancellation Points

Cancellation points shall occur when a thread is executing the following POSIX.1 or C Standard {2} functions:

aio_suspend()	*pause*()	*sigwait*()
close()	*pthread_cond_timedwait*()	*sigwaitinfo*()
creat()	*pthread_cond_wait*()	*sleep*()
fsync()	*pthread_join*()	*system*()

mq_receive()	*pthread_testcancel*()	*tcdrain*()
mq_send()	*read*()	*wait*()
`msync()`	`sem_wait()`	`waitpid()`
nanosleep()	*sigsuspend*()	*write*()
open()	*sigtimedwait*()	
fcntl()	(when the *cmd* argument is F_SETLKW)	

A cancellation point may also occur when a thread is executing the following POSIX.1 or C Standard {2} functions:

closedir()	*ftell*()	*getpwnam*()	*puts*()
ctermid()	*fwrite*()	*getpwnam_r*()	*readdir*()
fclose()	*getc*()	*getpwuid*()	*remove*()
fflush()	*getc_unlocked*()	*getpwuid_r*()	*rename*()
fgetc()	*getchar*()	*gets*()	*rewind*()
fgets()	*getchar_unlocked*()	*lseek*()	*rewinddir*()
fopen()	*getcwd*()	*opendir*()	*scanf*()
fprintf()	*getgrgid*()	*perror*()	*tmpfile*()
fputc()	*getgrgid_r*()	*printf*()	*tmpname*()
fputs()	*getgrnam*()	*putc*()	*ttyname*()
fread()	*getgrnam_r*()	*putc_unlocked*()	*ttyname_r*()
freopen()	*getlogin*()	*putchar*()	*ungetc*()
fscanf()	*getlogin_r*()	*putchar_unlocked*()	*unlink*()
fseek()			
fcntl()	for any value of the command argument		

An implementation shall not introduce cancellation points into any other POSIX.1 or C Standard {2} functions.

The side effects of acting upon a cancellation request while suspended during a call of a POSIX.1 function shall be 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 shall occur before any cancellation cleanup handlers are called.

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

18.1.3 Thread Cancellation Cleanup Handlers

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

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

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

A side effect of acting upon a cancellation request while in a condition variable wait is that the mutex is reacquired before calling the first cancellation cleanup handler. In addition, the thread shall no longer be 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 shall not exit via *longjmp*() or *siglongjmp*().

18.1.4 Async-Cancel Safety

The functions *pthread_cancel*(), *pthread_setcancelstate*(), and *pthread_setcanceltype*() shall be async-cancel safe.

No other functions in POSIX.1 or the C Standard {2} are required to be async-cancel safe.

18.2 Thread Cancellation Functions

18.2.1 Canceling Execution of a Thread

Function: *pthread_cancel*()

18.2.1.1 Synopsis

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
```

18.2.1.2 Description

If {_POSIX_THREADS} is defined:

> The *pthread_cancel*() function requests that *thread* be canceled. The cancelability state and type of the target thread determines when the cancellation takes effect. When the cancellation is acted on, the cancellation cleanup handlers for *thread* are called. When the last cancellation cleanup handler returns, the thread-specific data destructor functions shall be called for *thread*. When the last destructor function returns, *thread* shall be terminated.
> The cancellation processing in the target thread runs asynchronously with respect to the calling thread returning from *pthread_cancel*().

Otherwise:

> Either the implementation shall support the *pthread_cancel*() function as described above or the *pthread_cancel*() function shall not be provided.

18.2.1.3 Returns

If successful, the *pthread_cancel*() function shall return zero. Otherwise, an error number shall be returned to indicate the error.

18.2.1.4 Errors

For each of the following conditions, if the condition is detected, the *pthread_cancel*() function shall return the corresponding error number:

[ESRCH]	No thread could be found corresponding to that specified by the given thread ID.

18.2.1.5 Cross-References

pthread_exit(), 16.2.5; *pthread_join*(), 16.2.3; *pthread_setcancelstate*(), 18.2.2; *pthread_cond_wait*(), 11.4.4; *pthread_cond_timedwait*(), 11.4.4.

18.2.2 Setting Cancelability State

Functions: *pthread_setcancelstate*(), *pthread_setcanceltype*(), *pthread_testcancel*()

18.2.2.1 Synopsis

```
#include <pthread.h>
int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
void pthread_testcancel(void);
```

18.2.2.2 Description

If {_POSIX_THREADS} is defined:

> The *pthread_setcancelstate*() function atomically both sets the cancelability state of the calling thread to the indicated *state* and returns 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 atomically both sets the cancelability type of the calling thread to the indicated *type* and returns 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 creates a cancellation point in the calling thread. The *pthread_testcancel*() function has no effect if cancelability is disabled.

Otherwise:

> Either the implementation shall support the *pthread_setcancelstate*(), *pthread_setcanceltype*(), and *pthread_testcancel*() functions as described above or the *pthread_setcancelstate*(), *pthread_setcanceltype*(), and *pthread_testcancel*() functions shall not be provided.

18.2.2.3 Returns

If successful, the *pthread_setcancelstate*() and *pthread_setcanceltype*() functions shall return zero. Otherwise, an error number shall be returned to indicate the error.

18.2.2.4 Errors

For each of the following conditions, if the condition is detected, the *pthread_setcancelstate*() function shall return the corresponding error number:

[EINVAL]	The specified state is not PTHREAD_CANCEL_ENABLE or PTHREAD_CANCEL_DISABLE.

For each of the following conditions, if the condition is detected, the *pthread_setcanceltype*() function shall return the corresponding error number:

[EINVAL]	The specified type is not PTHREAD_CANCEL_DEFERRED or PTHREAD_CANCEL_-ASYNCHRONOUS.

18.2.3 Establishing Cancellation Handlers

Functions: *pthread_cleanup_push*(), *pthread_cleanup_pop*()

18.2.3.1 Synopsis

```
#include <pthread.h>
void pthread_cleanup_push(void (*routine) (void *), void *arg);
void pthread_cleanup_pop(int execute);
```

18.2.3.2 Description

If {_POSIX_THREADS} is defined:

The *pthread_cleanup_push*() function pushes the specified cancellation cleanup handler *routine* onto the cancellation cleanup stack of the calling thread. The cancellation cleanup handler shall be popped from the cancellation cleanup stack and invoked with the argument *arg* when the thread exits [that is, calls *pthread_exit*()], the thread acts upon a cancellation request, or the thread calls *pthread_cleanup_pop*() with a nonzero *execute* argument.
The *pthread_cleanup_pop*() function removes the routine at the top of the cancellation cleanup stack of the calling thread and optionally invokes it (if *execute* is nonzero).
In the C language, these functions may be implemented as macros and shall appear as statements and in pairs within the same lexical scope [that is, the *pthread_cleanup_push*() macro may be thought to expand to a token list whose first token is "{" with *pthread_cleanup_pop*() expanding to a token list whose last token is the corresponding "}"].
The effect of calling *longjmp*() or *siglongjmp*() is undefined if there have been any calls to *pthread_cleanup_push*() or *pthread_cleanup_pop*() made without the matching call since the jump buffer was filled. The effect of calling *longjmp*() or *siglongjmp*() from inside a cancellation cleanup handler is also undefined unless the jump buffer was also filled in the cancellation cleanup handler.

Otherwise:

Either the implementation shall support the *pthread_cleanup_push*() and *pthread_cleanup_pop*() functions as described above or the *pthread_cleanup_push*() and *pthread_cleanup-pop*() functions shall not be provided.

18.2.3.3 Returns

The *pthread_cleanup_push*() and *pthread_cleanup_pop*() functions shall be used as statements.

18.2.3.4 Errors

This standard does not specify any error conditions for the *pthread_cleanup_push*() or the *pthread_cleanup_pop*() functions.

18.2.3.5 Cross-References

pthread_cancel(), 18.2.1; *pthread_setcancelstate*(), 18.2.2.

18.3 Language-Independent Cancellation Functionality

This clause presents the language-independent thread cancellation functionality required of all language bindings.

18.3.1 Requesting Cancellation

All language bindings for thread cancellation shall provide a mechanism to request the cancellation of the computation currently being run by a thread. The C language binding for this mechanism is the *pthread_cancel*() function.

18.3.2 Associating Cleanup Code With Scopes

All language bindings for thread cancellation shall provide a mechanism for associating cleanup code (which is run when a scope of code is canceled) with that scope. The mechanism shall allow an arbitrary number of such scopes and their cleanup codes to be established and for such scopes to stack. The C language bindings for this mechanism are the *pthread_cleanup_push*() and *pthread_cleanup_pop*() routines.

18.3.3 Controlling Cancellation Within Scopes

All language bindings for thread cancellation shall provide a mechanism for controlling the points at which cancellation may occur within scopes. This mechanism shall be usable in a modular fashion such that the cancellation control choices made within one scope are not violated or invalidated by choices made by nested or called scopes. It is recommended that the default cancellation control environment permit cancellation only at defined cancellation points. The C language bindings for this mechanism are the *pthread_setcancelstate*(), *pthread_setcanceltype*(), and *pthread_testcancel*() functions.

18.3.4 Defined Cancellation Sequence

All language bindings for thread cancellation shall define the sequence of execution and the execution environment present during cancellation. All bindings are constrained to execute sets of scope cleanup code in the reverse order from the order in which the associated scopes were entered (i.e., in LIFO order). Bindings may provide a mechanism for cleanup code within the thread being canceled to indicate that the cancellation has been completed before all scopes are cleaned up, allowing execution to resume within the scope associated with the cleanup code. The C language binding provides no mechanism for completing a cancellation before all cleanup code has been run and the canceled thread has been terminated.

18.3.5 List of Cancellation Points

All language bindings for thread cancellation shall provide a complete list of the cancellation points for all functions provided with the language binding. It is intended that this list consist of those functions that may block for unbounded periods of time. This part of ISO/IEC 9945 provides such a list.

Annex A Bibliography

(Informative)

This Annex contains lists of related open systems standards and suggested reading on historical implementations and application programming.

A.1 Related Open Systems Standards

A.1.1 Networking Standards

{B1}ISO 7498: 1984, *Information processing systems—Open Systems Interconnection—Basic Reference Model*.[4]

{B2}ISO 8072: 1986, *Information processing systems—Open Systems Interconnection—Transport service definition.*

{B3}ISO/IEC 8073: 1988, *Information processing systems—Open Systems Interconnection—Connection oriented transport protocol specification*.[5]

{B4}ISO 8326: 1987, *Information processing systems—Open Systems Interconnection—Basic connection oriented session service definition*.

{B5}ISO 8327: 1987, *Information processing systems—Open Systems Interconnection—Basic connection oriented session protocol definition*.

{B6}ISO 8348: 1987, *Information processing systems—Data communications—Network service definition*.

{B7}ISO 8473: 1988, *Information processing systems—Data communications—Protocol for providing the connectionless-mode network service*.

{B8}ISO 8571: 1988, *Information processing systems—Open Systems Interconnection—File Transfer, Access and Management*.

{B9}ISO 8649: 1988, *Information processing systems—Open Systems Interconnection—Service definition for the Association Control Service Element.*

{B10}ISO 8650: 1988, *Information processing systems—Open Systems Interconnection—Protocol specification for the Association Control Service Element.*

{B11}ISO 8802-2:1989 [IEEE Std 802.2-1989 (ANSI)], *Information processing systems—Local area networks—Part 2: Logical link control.*

{B12}ISO 8802-3:1989 [IEEE Std 802.3-1988 (ANSI)], *Information processing systems—Local area networks—Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications.*

{B13}ISO/IEC 8802-4:1990 [IEEE Std 802.4-1990 (ANSI)], *Information technology—Local area networks—Part 4: Token-passing bus access method and physical layer specifications.*

{B14}ISO 8802-5:… (IEEE 802.5-1989), *Information technology—Local area networks—Part 5: Token ring access method and physical layer specifications.*

[4]ISO documents can be obtained from the ISO office, 1, rue de Varembé, Case Postale 56, CH-1211, Genève 20, Switzerland/Suisse.
[5]IEC documents can be obtained from the IEC office, 3, rue de Varembé, Case Postale 131, CH-1211, Genève 20, Switzerland/Suisse.

{B15}ISO 8822: 1988, *Information processing systems—Open Systems Interconnection—Connection oriented presentation service definition.*

{B16}ISO 8823: 1988, *Information processing systems—Open Systems Interconnection—Connection oriented presentation protocol specification.*

{B17}ISO 8831: 1989, *Information processing systems—Open Systems Interconnection—Job transfer and manipulation concepts and services.*

{B18}ISO 8832: 1989, *Information processing systems—Open Systems Interconnection—Specification of the basic class protocol for job transfer and manipulation.*

{B19}CCITT Recommendation X.25, *Interface between data terminal equipment (DTE) and data circuit-terminating equipment (DCT) for terminals operating in the packet mode and connected to public data networks by dedicated circuit.*[6]

{B20}CCITT Recommendation X. 212, *Information processing systems—Data communication—Data link service definition for Open Systems Interconnection.*

A.1.2 Language Standards

{B21}ISO 1539: 1980, *Programming languages—FORTRAN.*

{B22}ISO 1989: 1985, *Programming Languages—COBOL.*

{B23}ISO 8652: 1987, *Programming Languages—Ada.*

{B24}ANSI X3.113-1987[7], *Information systems—Programming language—FULL BASIC.*

{B25}ANSI/IEEE 770X3.97-1983, *Standard Pascal Computer Programming Language.*

{B26}ANSI/MDC X11.1-1984, *Programming Language MUMPS.*

A.1.3 Graphics Standards

{B27}ISO 7942: 1985, *Information processing systems—Computer graphics—Graphical Kernel System (GKS) functional description.*

{B28}ISO 8632: 1987, *Information processing systems—Computer graphics—Metafile for the storage and transfer of picture description information.*

{B29}ISO/IEC 9592: 1989 (ANSI X3.144-1988), *Information processing systems—Computer graphics—Programmer's hierarchical interactive graphics system (PHIGS).*

A.1.4 Database Standards

{B30}ISO 8907: 1987, *Database Language—NDL.*

{B31}ISO 9075: 1987, *Database Language—SQL.*

[6]CCITT documents can be obtained from the International Telecommunications Union, Sales Section, Place des Nations, CH-1211, Genève 20, Switzerland/Suisse.
[7]ANSI documents can be obtained from the Sales Department, American National Standards Institute, 1430 Broadway, New York, NY 10018.

A.2 Other Standards

{B32}ISO 639: 1988, *Code for the representation of names of languages.*

{B33}ISO 3166: 1988, *Code for the representation of names of countries.*

{B34}ISO 8859-1: 1987, *Information Processing—8-bit single-byte coded graphic character sets—Part 1: Latin alphabet No. 1.*

{B35}ISO 9127: 1988, *Information processing systems—User documentation and cover information for consumer software packages.*

{B36}ISO/IEC 9945-2:…,, [8] *Information technology—Portable operating system interface (POSIX)—Part 2: Shell and utilities.*

{B37}ISO/IEC 10646:…,, [9] *Information processing—Multiple octet coded character set.*

{B38}IEEE Std 100-1988, *IEEE Standard Dictionary of Electrical and Electronics Terms.*

{B39}P1003.0/D16,[10] *Draft Guide to the POSIX Open Systems Environment.*

{B40}ISO/IEC TR 10000-1: 1990, *Information technology—Framework and taxonomy of International Standardized Profiles—Part 1: Framework.*

A.3 Historical Documentation and Introductory Texts

{B41}American Telephone and Telegraph Company. *System V Interface Definition (SVID), Issues 2 and 3.* Morristown, NJ: UNIX Press, 1986, 1989.[11]

{B42}American Telephone and Telegraph Company. *UNIX System III Programmer's Manual.* Greensboro, NC: Western Electric Company, October 1981.

{B43}American Telephone and Telegraph Company. *UNIX Time Sharing System: UNIX Programmer's Manual.* 7th ed. Murray Hill, NJ: Bell Telephone Laboratories, January 1979.

{B44}Apollo Computer, Inc., *Concurrent Programming Support (CPS) Reference*, Order No. 010233, June 1987.

{B45}"The UNIX System."[12] *AT&T Bell Laboratories Technical Journal.* vol. 63 (8 Part 2), October 1984.

{B46}"UNIX Time-Sharing System."[13] *Bell System Technical Journal.* vol. 57 (6 Part 2), July–August 1978.

{B47}Bach, Maurice J. *The Design of the UNIX Operating System.* Englewood Cliffs, NJ: Prentice-Hall, 1987.

{B48}Birrell, A., *An Introduction to Programming with Threads*, DEC SRC Research Report 35, DEC Systems Research Center, 130 Lytton Ave., Palo Alto, CA, January 1989.

[8]To be approved and published.
[9]To be approved and published.
[10]This unapproved draft document is available from IEEE Publications, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331. Telephone: 1 (800) 678-IEEE or +1 (908)981-1393 (outside US).
[11]This is one of several documents that represent an industry specification in an area related to POSIX.1. The creators of such documents may be able to identify newer versions that may be interesting.
[12]This entire edition is devoted to the UNIX system.
[13]This entire edition is devoted to the UNIX time-sharing system.

{B49}Cooper, E. and Draves, R., *C Threads*, Technical Report CMU-CS-88-154, Carnegie Mellon University, Computer Science Department, Pittsburgh, PA, June 1988.

{B50}Dijkstra, E. W. "Solution of a Problem in Concurrent Programming Control," *Communications of the ACM.* vol. 8 (9), September 1965, pp. 569–570.

{B51}Digital Equipment Corporation, *Concert Multi-thread(TM) Architecture*, Revision 1.0–2, April 1989.

{B52}Doeppner, T., *Threads: A system for the support of concurrent programming*, Technical Report CS-87-11, Brown University, Computer Science Department, Providence, RI, June 1987.

{B53}Furht, Borko, Grostick, Dan, Gluch, David, Rabbat, Guy, Parker, John, and McRoberts, Meg. *Real-Time UNIX Systems: Design and Application Guide.* Boston, MA: Kluwer Academic Publishers, 1991.

{B54}Harbison, Samuel P. and Steele, Guy L. *C: A Reference Manual.* Englewood Cliffs, NJ: Prentice-Hall, 1987.

{B55}Hoare, C.A.R., "Monitors: An operating system structuring concept," *Communications of the ACM*, vol. 17, no. 10, pp. 549–557, October 1974.

{B56}Jones, Michael B., "Bringing the C Libraries With Us into a Multi-Threaded Future," *Winter 1991 Usenix Conference Proceedings*, Dallas, TX, pp. 81–91, January 1991.

{B57}Kernighan, Brian W. and Ritchie, Dennis M. *The C Programming Language.* Englewood Cliffs, NJ: Prentice-Hall, 1978.

{B58}Kernighan, Brian W. and Pike, Rob. *The UNIX Programming Environment.* Englewood Cliffs, NJ: Prentice-Hall, 1984.

{B59}Lampson, B. and Redell, D., "Experience with processes and monitors in Mesa," *Communications of the ACM*, vol. 23, no. 2, pp. 105–117, February 1980.

{B60}Leffler, Samuel J., McKusick, Marshall Kirk, Karels, Michael J., Quarterman, John S., and Stettner, Armando. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Reading, MA: Addison-Wesley, 1988.

{B61}McGilton, Henry and Morgan, Rachel. *Introducing the UNIX System.* New York: McGraw-Hill (BYTE Books), 1983.

{B62}McJones, P. and Swart, G., "Evolving the UNIX system interface to support multi-threaded programs," *Proceedings of the Winter 1989 USENIX Conference*, pp. 393–404, February 1989.

{B63}Organick, Elliot I. *The Multics System: An Examination of Its Structure.* Cambridge, MA: The MIT Press, 1972.

{B64}Quarterman, John S., Silberschatz, Abraham, and Peterson, James L. "4.2BSD and 4.3BSD as Examples of the UNIX System." *ACM Computing Surveys.* vol. 17 (4), December 1985, pp. 379–418.

{B65}Ritchie, Dennis M. "Reflections on Software Research." *Communications of the ACM.* vol. 27 (8), August 1984, pp. 758–760. ACM Turing Award Lecture.

{B66}Ritchie, Dennis. "The Evolution of the UNIX Time-Sharing System." *AT&T Bell Laboratories Technical Journal.* vol. 63 (8), October 1984, pp. 1577–1593.

{B67}Ritchie, D. M. and Thompson, K. "The UNIX Time-Sharing System." *Communications of the ACM.* vol. 7 (7), July 1974, pp. 365–375. This is the original paper, which describes Version 6.

{B68}Ritchie, D. M. and Thompson, K. “The UNIX Time-Sharing System.” *Bell System Technical Journal.* vol. 57 (6 Part 2), July-August 1978, pp, 1905–1929. This is a revised version and describes Version 7.

{B69}Ritchie, Dennis M. “Unix: A Dialectic.” *Winter 1987 USENIX Association Conference Proceedings, Washington, D.C.,* pp. 29–34. Berkeley, CA: USENIX Association, January 1987.

{B70}Sun Microsystems Inc., *Sun OS 4.0 Reference Manual*, Chapter 6, May 1988.

{B71}Rochkind, Marc J. *Advanced UNIX Programming.* Englewood Cliffs, NJ: Prentice-Hall, 1985.

{B72}Tucker, Andrew and Gupta, Anoop. “Process Control and Scheduling Issues for Multipgrammed Shared-Memory Processors.” *Proceedings of the 12th ACM Symposium on Operating System Principles*, pp. 159–166. December, 1989.

{B73}University of California at Berkeley—Computer Science Research Group. *4.3 Berkeley Software Distribution, Virtual VAX-11 Version.* Berkeley, CA: The Regents of the University of California, April 1986.

{B74}UNIX International Inc., *Multiprocessor Working Group Report*, 1990.

{B75}/usr/group Standards Committee. *1984 /usr/group Standard.* Santa Clara, CA: UniForum, 1984.

{B76}X/Open Company, Ltd. *X/Open Portability Guide, Issue 2.* Amsterdam: Elsevier Science Publishers, 1987.

{B77}X/Open Company, Ltd. *X/Open Portability Guide, Issue 3.* Englewood Cliffs, NJ: Prentice-Hall, 1989.

A.4 Other Sources of Information

[B78] ISO/IEC JTC 1 N1335, *Final Report of ISO/IEC JTC 1 TSG-1 on Standards Necessary to Define Interfaces for Application Portability (IAP).*

Annex B Rationale and Notes

(Informative)

The annex is being published as an informative part of POSIX.1 to assist in the process of review. It contains historical information concerning the contents of POSIX.1 and why features were included or discarded. It also contains notes of interest to application programmers on recommended programming practices, emphasizing the consequences of some aspects of POSIX.1 that may not be immediately apparent.[14]

B.1 Scope and Normative Cross-References

B.1.1 Scope

This rationale focuses primarily on additions, clarifications, and changes made to the UNIX system, from which POSIX.1 was derived. It is not a rationale for the UNIX system as a whole, since the goal of the developers of POSIX.1 was to codify existing practice, not design a new operating system. No attempt is made in this rationale to defend the pre-existing structure of UNIX systems. It is primarily deviations from existing practice, as codified in the base documents, that are explained or justified here.

Material that is "outside the scope" or otherwise not addressed by this part of ISO/IEC 9945 is implicitly "unspecified." It may be included in an implementation, and thus the implementation does provide a specification for it. The term "implementation defined" has a specific meaning in POSIX.1 and is not a synonym for "defined (or specified) by the implementation."

The rationale discusses some UNIX system features that were *not* adopted into POSIX.1. Many of these are features that are popular in some UNIX system implementations, so that a user of those implementations might question why they do not appear in POSIX.1. This rationale should provide the appropriate answers.

There are choices allowed by POSIX.1 for some details of the interface specification; some of these are specifiable optional subsets of POSIX.1. See B.2.9.

Although the services POSIX.1 provides have been defined in the C language, the concept of providing fundamental, standardized services should not be restricted only to programs of a particular programming language. The possibility of implementing interfaces in alternate programming languages inspired the term *POSIX.1 with a C Language Binding.* The word *Binding* refers to the binding of a conceptual set of services and a standardized C interface that establishes rules and syntax for accessing them. Future international standards are expected to separate the C language binding from the language-independent services of POSIX.1 and to include bindings for other programming languages.

The C Standard {2} will be the basis for functional definitions of core services that are independent of programming languages. POSIX.1 as it stands now can be thought of as a C Language Binding. Sections 1 through 7, and 9, correspond roughly to the C language implementation of what will be defined in the programming language-independent core services portion of POSIX.1; Section 8 corresponds to the C language-specific portion.

The criteria used to choose the programming language-independent core services may be different from those expected. The core services represent services that are common to those programming languages likely to form language bindings to POSIX.1—the greatest common denominator. They are not chosen to reflect the most important system services of an ideal operating system. For this reason, some fundamental system services are not included in the language-independent core. As an example, memory management routines would at first seem to be a core

[14]The material in this annex is derived in part from copyrighted draft documents developed under the sponsorship of UniForum, as part of an ongoing program of that association to support the POSIX standards program efforts.

service—they are an absolutely fundamental system service. They must, however, be included in language-specific portions of POSIX.1 because programming languages such as FORTRAN have traditionally not provided memory management. Categorizing memory management as a core service would impose unreasonable requirements for FORTRAN implementations.

Any programming language traditionally supporting memory management should include those routines in the language-dependent portions of their bindings. Work will be done at a later time to standardize the classes of functions that must be included in the language-dependent portions of language bindings if those functions have been traditionally implemented for that language. This will ensure that certain classes of critical functions, such as memory management, will not be excluded from any applicable language binding; see B.1.3.3.

POSIX.1 is not a tutorial on the use of the specified interface, nor is this rationale. However, this part of ISO/IEC 9945 includes a bibliography of well-regarded historical documentation on the UNIX system in A.3.

B.1.1.1 POSIX.1 and the C Standard

Some C language functions and definitions were handled by POSIX.1, but most were handled by the C Standard {2}. The general guideline is that POSIX.1 retained responsibility for operating-system specific functions, while the C Standard {2} defined C library functions. See also B.2.7 and B.8.

There are several areas in which the two standards differ philosophically:

1) *Function parameter type lists.* These appear in the syntax of the C Standard {2}. In this version of POSIX.1, the parameter lists were restated in terms of these function prototypes. There were two major reasons for making this change from IEEE Std 1003.1-1988 : the use of the C Standard {2} was rapidly becoming more widespread, and implementors were experiencing difficulties with some of the function prototypes where guidance was not provided in POSIX.1. (The modifier `const` provided the most difficulty.) Specific guidance and permission remains in POSIX.1 for translation to common-usage C.
2) *Single vs. multiple processes.* The C Standard {2} specifies a language that can be used on single-process operating systems and as a freestanding base for the implementation of operating systems or other stand alone programs. However, the POSIX.1 interface is that of a multiprocess timesharing system. Thus, POSIX.1 has to take multiple processes into account in places where the C Standard {2} does not mention processes at all, such as *kill*(). See also B.3.3.1.1B.1.3.1.1.
3) *Single vs. multiple operating system environments.* The C Standard {2} specifies a language that may be useful on more than one operating system and thus has means of tailoring itself to the particular current environment. POSIX.1 is an operating system interface specification and thus by definition is only concerned with one operating system environment, even though it has been carefully written to be broadly implementable (see Broadly Implementable in the Introduction) in terms of various underlying operating systems. See also B.1.3.1.1.
4) *Translation vs. execution environment.* POSIX.1 is primarily concerned with the C Standard {2} *execution environment*, leaving the *translation environment* to the C Standard {2}. See also B.1.3.1.1.
5) *Hosted vs. freestanding implementations.* All POSIX.1 implementations are hosted in the sense of the C Standard {2}. See also the remarks on conformance in the Introduction.
6) *Text vs. binary file modes.* The C Standard {2} defines *text* and *binary* modes for a file. But the POSIX.1 interface and historical implementations related to it make no such distinction, and all functions defined by POSIX.1 treat files as if these modes were identical. (It should not be stated that POSIX.1 files are either *text* or *binary*.) The definitions in the C Standard {2} were written so that this interpretation is possible. In particular, *text* mode files are not required to end with a line separator, which also means that they are not required to include a line separator at all.

Furthermore, there is a basic difference in approach between the Rationale accompanying the C Standard {2} and this Rationale Annex. The C Standard {2} Rationale, a separate document, addresses almost all changes as differences from the Base Documents of the C Standard {2}, usually either Kernighan and Ritchie {B57} or the *1984 /usr/group Standard* {B75} . This Rationale cannot do that, since there are many more variants of (and Base Documents for) the operating system interface than for the C language. The most noticeable aspect of this difference is that the C Standard {2} Rationale identifies "QUIET CHANGES" from the Base Documents. This Annex cannot include such markings, since a quiet change from one historical implementation may correspond exactly to another historical implementation, and may be very noticeable to an application written for yet another.

The following subclauses justify the inclusion or omission of various C language functions in POSIX.1 or the C Standard {2}.

B.1.1.1.1 Solely by POSIX.1

These return parameters from the operating system environment: *ctermid*(), *ttyname*(), and *isatty*().

The *fileno*() and *fdopen*() functions map between C language stream pointers and POSIX.1 file descriptors.

B.1.1.1.2 Solely by the C Standard

There are many functions that are useful with the operating system interface and are required for conformance with POSIX.1, but that are properly part of the C Language. These are listed in 8.1, which also notes which functions are defined by both POSIX.1 and the C Standard {2}. Certain terms defined by the C Standard {2} are incorporated by POSIX.1 in 2.7.

Some routines were considered too specialized to be included in POSIX.1. These include *bsearch*() and *qsort*().

B.1.1.1.3 By Neither POSIX.1 Nor the C Standard

Some functions were considered of marginal utility and problematical when international character sets were considered: *_toupper*(), *_tolower*(), *toascii*(), and *isascii*().

Although *malloc*() and *free*() are in the C Standard {2} and are required by 8.1 of POSIX.1, neither *brk*() nor *sbrk*() occur in either standard (although they were in the *1984* / *usr* / *group Standard* {B75}), because POSIX.1 is designed to provide the basic set of functions required to write a Conforming POSIX.1 Application; the underlying implementation of *malloc*() or *free*() is not an appropriate concern for POSIX.1.

B.1.1.1.4 Base by POSIX.1, Additions by the C Standard

Since the C Standard {2} does not depend on POSIX.1 in any way, there are no items in this category.

B.1.1.1.5 Base by the C Standard, Additions by POSIX.1

The C Standard {2} has to define *errno* if only because examining that variable offers the only way to determine when some mathematics routines fail. But POSIX.1 uses it more extensively and adds some semantics to it in 2.4, which also defines some values for it.

Many numerical limits used by the C Standard {2} were incorporated by POSIX.1 in 2.8, and some new ones were added, all to be found in the header `<limits.h>`.

The C Standard {2} provides *signal*(), a minimal functionality for interrupts. The POSIX.1 definition replaces this with an elaborate mechanism that deals with multiple processes and is reliable when signals come from outside sources.

The *time*() function is used by the C Standard {2}, but POSIX.1 further specifies the time value.

The *getenv*() function is referenced in 2.6 and 3.1.2 and is also defined by the C Standard {2}.

The *rename*() function is extended to further specify its behavior when the new filename already exists or either argument refers to a directory.

The *setlocale*() function and the handling of time zones were further specified to take advantage of the POSIX environment.

The standard-I/O functions were specified in terms of their relationship to file descriptors and the relationship between multiple processes.

B.1.1.1.6 Related Functions by Both

The C Standard {2} definition of *compliance* and the POSIX.1 definition of *conformance* are similar, although the latter notes certain potential hardware limitations.

POSIX.1 defined a portable filename character set in 2.2.2 that is like the C Standard {2} identifier character set. However, POSIX.1 did not allow upper and lowercase characters to be considered equivalent. See *filename portability* in 2.3.4.

The *exit*() function is defined only by the C Standard {2} because it refers to closing streams, and that subject, as well as *fclose*() itself, is defined almost entirely by the C Standard {2}. But POSIX.1 defined *_exit*(), which also adds semantics to *exit*(). This allows POSIX.1 to omit references to the C Standard {2} *atexit*() function.

POSIX.1 defined *kill*(), while the C Standard {2} defined *raise*(), which is similar except that it does not have a process ID argument, since the language defined by the C Standard {2} does not incorporate the idea of multiple processes.

The new functions *sigsetjmp*() and *siglongjmp*() were added to provide similar functions to the C Standard {2} *setjmp*() and *longjmp*() that additionally save and restore signal state.

B.1.1.2 Threads

Threads are an emerging model for expressing parallelism within a process in POSIX.

On a system that provides hardware and software support for parallel execution, the use of threads can increase the speed of execution by providing an application programmer with the ability to utilize all the available processors simultaneously.

Even on a uniprocessor system, threads are useful for mapping asynchronous behavior into equivalent synchronous behavior such as providing I/O parallelism, controlling asynchronous computations, or structuring applications composed of many logically distinct tasks (e.g., simulations and windowing systems).

The thread model is based on a well-understood synchronous, procedural model consistent with the C function model. Threads can be used to model the parallelism inherent in windowing environments, realtime event processing, Ada tasking, transaction processing, and networked/distributed systems.

It is required that a threads standard provide for the following:

- Preservation of POSIX.1 syntax and semantics relative to threads.
- Support for a model for explicit parallelism within a process. This requires support for the explicit creation and termination of multiple schedulable threads of control within the address space of a process and requires that a suspended thread not suspend other threads within the same process.
- Support for various models of thread scheduling. Unless the system is a multiprocessor, these shall include deterministic scheduling in a manner equivalent to the scheduling options available under this part of ISO/IEC 9945.

— Support for the efficient synchronization of access to objects within the process address space by multiple threads; for example, access to critical sections.
— Support for a facility that allows for one or more threads to wait for the occurrence of some condition internal to the process; for example, waiting for the completion of a computation.
— Support for a mechanism that allows threads to maintain thread-specific data.

The design of this facility involved compromises between compatibility with existing practice and the optimal solution to a problem. The philosophy adopted was to aim for a high level of compatibility and to break with existing practice only if absolutely necessary.

B.1.2 Normative Cross-References

There is no additional rationale provided for this subclause.

B.1.3 Conformance

These conformance definitions are descended from those of *conforming implementation*, *conforming application*, and *conforming portable application* of early drafts, but were changed to clarify

1) Extensions, options, and limits;
2) Relations among the three terms, and;
3) Relations between POSIX.1 and the C Standard {2}.

B.1.3.1 Implementation Conformance

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

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

B.1.3.1.1 Requirements

The word "support" is used, rather than "provide," in order to allow an implementation that has no resident software development facilities, but that supports the execution of a *Strictly Conforming POSIX.1 Application*, to be a *conforming implementation*. See also B.1.1.1.

B.1.3.1.2 Documentation

The conforming documentation is required to use the same numbering scheme as POSIX.1 for purposes of cross referencing. This requirement is consistent with and supplements the verification test assertions being developed by other POSIX groups. All options that an implementation chooses shall be reflected in `<limits.h>` and `<unistd.h>`.

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

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

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

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

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

B.1.3.1.3 Conforming Implementation Options

Within POSIX.1 there are some symbolic constants that, if defined, indicate that a certain option is enabled. Other symbolic constants exist in POSIX.1 for other reasons. This clause helps clarify which constants are related to true "options" and which are related more to the behavior of differing systems.

To accommodate historical implementations where there were distinct semantics in certain situations, but where one was not clearly better or worse than another, early drafts of POSIX.1 permitted either of (typically) two options using "may." At the request of the working group developing test assertions, this was changed to be specified by formal options with flags. It quickly became obvious that these would be treated as options that could be selected by a purchaser, when the intent of the developers of POSIX.1 was to allow either behavior (or both, in some cases) to conform to the standard, and to constrain the application to accommodate either. Thus, these options were removed and the phrase "An implementation may either" introduced to replace the option. Where this phrase is used, it indicates that an application shall tolerate either behavior.

It is intended that all conforming applications shall tolerate either behavior and that only in the most exceptional of circumstances (driven by technical need) should a purchaser specify only one behavior. Backwards compatibility is not considered exceptional, as this is not consistent with the intent of POSIX.1: to promote the portability of applications (and the development of portable applications).

An application can tolerate these behaviors either by ignoring the differences (if they are irrelevant to the application) or by taking an action to assure a known state. It might be that that action would be redundant on some implementations.

Validation programs, which are applications in this sense, could either report the actual result found or simply ignore the difference. In no case should either acceptable behavior be treated as an error. This may complicate the validation slightly, but is more consistent with the intent of this permissible variation in behavior.

In certain circumstances, the behavior may vary for a given process. For example, in the presence of networked file systems, whether or not dot and dot-dot are present in the directory may vary with the directory being searched, and the program would only be portable if it tolerated, but did not require, the presence of these entries in a directory.

In situations like this, it is typically easier to simply ignore dot and dot-dot if they are found than to try to determine if they should be expected or not.

B.1.3.2 Application Conformance

These definitions guide users or adaptors of applications in determining on which implementations an application will run and how much adaptation would be required to make it run on others. These three definitions are modeled after related ones in the C Standard {2}.

POSIX.1 occasionally uses the expressions *portable application* or *conforming application*. As they are used, these are synonyms for any of these three terms. The differences between the three classes of application conformance relate to the requirements for other standards, or, in the case of the Conforming POSIX.1 Application Using Extensions, to implementation extensions. When one of the less explicit expressions is used, it should be apparent from the context of the discussion which of the more explicit names is appropriate.

B.1.3.2.1 Strictly Conforming POSIX.1 Application

This definition is analogous to that of a C Standard {2} *conforming program*.

The major difference between a *Strictly Conforming POSIX.1 Application* and a C Standard {2} *strictly conforming program* is that the latter is not allowed to use features of POSIX.1 that are not in the C Standard {2}.

B.1.3.2.2 Conforming POSIX.1 Application

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

B.1.3.2.3 Conforming POSIX.1 Application Using Extensions

Due to possible requirements for configuration or implementation characteristics in excess of the specifications in 2.8.2 or related to the hardware (such as array size or file space), not every Conforming POSIX.1 Application Using Extensions will run on every conforming implementation.

B.1.3.3 Language-Dependent Services for the C Programming Language

POSIX.1 is, for historical reasons, both a specification of an operating system interface and a C binding for that specification. It is clear that these need to be separated into unique entities, but the urgency of getting the initial standard out, and the fact that C is the *de facto* primary language on systems similar to the UNIX system, makes this a necessary and workable situation.

Nevertheless, work will be done on language bindings, beyond that for C before the specification and the current binding are separated. Language bindings for languages other than C should not model themselves too closely on the C binding and in the process pick up various idiosyncrasies of C.

Where functionality is duplicated in POSIX.1 [e.g., *open*() and *creat*()] there is no reason for that duplication to be carried forward into another language. On the other hand, some languages have functionality already in them that is essentially the same as that provided in POSIX.1. In this case, a mapping between the functionality in that language and the underlying functionality in POSIX.1 is a better choice than mimicking the C binding.

Since C has no syntax for I/O, and I/O is a large fraction of POSIX.1, the paradigm of functions has been used. This may not be appropriate to another language. For example, FORTRAN's REWIND statement is a candidate to map onto a special case of *lseek*(), and its SEEK statement may completely cover for *lseek*(). If this is the case, there is no reason to provide SUBROUTINEs with the same functionality. In the more general case, file descriptors and FORTRAN's logical unit numbers may have a useful mapping. FORTRAN's ERR= option in I/O operations might replace returning –1; the whole concept of errors might be handled differently.

As was done with C, it is not unreasonable for other language bindings to specify some areas that are undefined or unspecified by the underlying language standard or that are permissible as extensions. This may, in fact, solve some difficult problems.

Using as much as possible of the target language in the binding enhances portability. If a program wishes to use some POSIX.1 capabilities, and these are bound to the language statements rather than appearing as additional procedure or function calls, and the program does in fact conform to the language standard while using those functions, it will port to a larger range of systems than one that is obligated to use procedure or function calls introduced specifically for the binding to POSIX.1 to do the same thing.

A program that requires the POSIX.1 capabilities that are not bound to the standard language directly (as above) has no chance to be portable outside the POSIX.1 environment. It does not matter whether the extension is syntactic or a new function; it still will not port without effort. Given this, it seems unreasonable not to consider language extensions when determining how best to map the functionality of POSIX.1 into a particular language binding. For example, a new statement similar to READ, which loads the values from a call like *stat*(), might be the best solution for reading the data lists returned as structures in C into a list of FORTRAN variables.

No attempt to mimic *printf*() or *scanf*() (or the rest of the C Standard {2} functions) should be made; the equivalent functions in the language should be used. (Formatted READ and WRITE in FORTRAN, `read/readln` and `write/writeln` in Pascal, for example.)

There is an inherent special relationship between an operating system standard and a language standard. It is unlikely that standards for other kinds of features (such as graphics) will bind directly to statements in a general purpose language. However, an operating system standard should provide the services required by a language. This is an unusual situation, and the tendency to use only new functions and procedures when creating a binding should be examined carefully. (A one-to-one binding in all cases is probably not possible, but bindings such as those for standard I/O in Section 8 may be possible.)

Binding directly to the language, where possible, should be encouraged both by making maximal use of the mapping between the operating system and the language that naturally exists and, where appropriate, by having the languages request changes to the operating system to facilitate such a mapping. (A future inclusion of a truncate function, specifically for the FORTRAN ENDFILE statement, but that is also generally useful, is a good example.)

Part of the job of creating a binding is choosing names for functions that are introduced, and these will need to be appropriate for that language. It is possible to use other than the most restrictive form of a name, since, as discussed previously, using these functions inherently makes the application not portable to systems that are not POSIX.1, and if POSIX.1 conformant systems typically accept names that the lowest-common-denominator system will not, there is no reason to *a priori* exclude such names. (The specific example is C, where it is typically “non-UNIX” systems that limit external identifiers to six characters.)

See B.1.1 for additional information about C bindings.

B.1.3.3.1 Types of Conformance

There is no additional rationale provided for this subclause.

B.1.3.3.2 C Standard Language-Dependent System Support

The issue of “namespace pollution” needs to be understood in this context. See B.2.7.2.

B.1.3.3.3 Common-Usage C Language-Dependent System Support

The issue of “namespace pollution” needs to be understood in this context. See B.2.7.2.

B.1.3.4 Other C Language-Related Specifications

The information concerning the use of library functions was adapted from a description in the C Standard {2}. Here is an example of how an application program can protect itself from library functions that may or may not be macros, rather than true functions:

The *atoi* () function may be used in any of several ways:

1) By use of its associated header (possibly generating a macro expansion)
```
     #include <stdlib.h>
     /* ... */
     i = atoi (str);
```
2) By use of its associated header (assuredly generating a true function call)
```
     #include <stdlib.h>
     #undef atoi
     /* ... */
     i = atoi (str);
     or
     #include <stdlib.h>
     /* ... */
     i = (atoi) (str);
```
3) By explicit declaration
```
     extern int atoi (const char *);
     /* ... */
     i = atoi (str);
```
4) By implicit declaration
```
     /* ... */
     i = atoi (str);
```
   (Assuming no function prototype is in scope. This is not allowed by the C Standard {2} for functions with variable arguments; furthermore, parameter type conversion "widening" is subject to different rules in this case.)

Note that the C Standard {2} reserves names starting with `'_'` for the compiler. Therefore, the compiler could, for example, implement an intrinsic, built-in function *_asm_builtin_atoi*(), which it recognized and expanded into inline assembly code. Then, in `<stdlib.h>`, there could be the following:

```
     #define atoi (X) _asm_builtin_atoi (X)
```

The user's "normal" call to *atoi*() would then be expanded inline, but the implementor would also be required to provide a callable function named *atoi*() for use when the application requires it; for example, if its address is to be stored in a function pointer variable.

B.1.3.5 Other Language-Related Specifications

It is intended that "long" identifiers and multicase linkage would be supported on POSIX.1 systems for all languages, including C. This is where that condition is stated. The portion of the sentence about "if such extensions are" is included to permit languages that have an absolute maximum, or an absolute requirement of case folding, to be conformant.

The requirement for longer names is included for several reasons:

1) Most systems similar to POSIX.1 are already conformant.

2) Many existing language standards restrict the length of names to accommodate existing systems that cannot be modified to allow longer names. However, those systems are not expected to be POSIX.1 conformant, for other reasons.
3) Many historical applications rely on such long names.
4) Future languages (such as FORTRAN 88) are likely to require it.

Specific to FORTRAN 77 {B21} , that standard permits long names, and this part of ISO/IEC 9945 requires that FORTRAN implementations running on POSIX.1 support long names. The requirements of case distinction and length are considered orthogonal, but both are required if both are permitted by the language. Note that a language can be conformant to POSIX.1 even though a binding does not exist, because an application need not step outside the language standard to write a useful program.

This requirement permits the use of reasonable-length names in a POSIX.1 binding to a language such as FORTRAN. Clearly nothing prohibits a program that does conform to the FORTRAN minima to compile and run on POSIX.1.

It is within the constraints of POSIX.1 to specify the behavior of the language processors and linker, consistent with the language, as it is a specification for an execution environment. This is different than a package such as GKS {B27} , which can reasonably be expected to be ported to a system that enforces the language minima.

It might be argued that this specification is appropriate to the language binding committees for POSIX generally, rather than specifically to POSIX.1. That argument misses the intent. The intent is to require that the linker and other code that handles "object code" (a concept not formally defined in POSIX.1) are able to support long names. This requirement, being one that spans all languages, belongs in the specification standard, rather than tied to any one language. Note that it is also somewhat permissive, in that if the language is unable to deal with long names it is permitted not to require them, but it does remove the argument that "the loader might not permit long names, so [a specific] language binding should not force the issue."

A strictly conforming application for a given language could not use any extensions outside of POSIX.1 for that language (regardless of the underlying operating system). An application will strictly conform to POSIX.1 if it conforms to the language using additional interfaces from that language's binding to POSIX.1.

B.2 Definitions and General Requirements

B.2.1 Conventions

There is no additional rationale provided for this subclause.

B.2.2 Definitions

B.2.2.1 Terminology

The meanings specified in POSIX.1 for the words *shall*, *should*, and *may* are mandated by ISO/IEC directives.

In this Rationale, the words *shall*, *should*, and *may* are sometimes used to illustrate similar usages in the standard. However, the Rationale itself does not specify anything regarding implementations or applications.

conformance document: As a practical matter, the conformance document is effectively part of the system documentation. They are distinguished by POSIX.1 so that they can be referred to distinctly.

implementation defined: This definition is analogous to that of the C Standard {2} and, together with *undefined* and *unspecified*, provides a range of specification of freedom allowed to the interface implementor.

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

shall: Declarative sentences are sometimes used in POSIX.1 as if they included the word *shall*, and facilities thus specified are no less required. For example, the two statements:

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

are meant to be exactly equivalent. It is expected that a future version of POSIX.1 will be rewritten to use the "shall" form more consistently.

should: In POSIX.1, the word *should* does not usually apply to the implementation, but rather to the application. Thus, the important words regarding implementations are *shall*, which indicates requirements, and *may*, which indicates options.

obsolescent: The term *obsolescent* was preferred over *deprecated* to represent functionality that should not be used in new work. The term *obsolescent* is more intuitive and reduced the possibility of misunderstanding in the intended context.

supported: An example of this concept is the *setpgid*() function. If the implementation does not support the job control feature provided under the Job Control option, it nevertheless has to provide a function named *setpgid*(), even though its only ability is that of returning [ENOSYS].

system documentation: The system documentation should normally describe the whole of the implementation, including any extensions provided by the implementation. Such documents normally contain information at least as detailed as the POSIX.1 specifications. Few requirements are made on the system documentation, but the term is needed to avoid a dangling pointer where the conformance document is permitted to point to the system documentation.

undefined: See *implementation defined*.

unspecified: See *implementation defined*.

The definitions for *unspecified* and *undefined* appear nearly identical at first examination, but are not. *Unspecified* means that a conforming program may deal with the unspecified behavior, and it should not care what the outcome is. *Undefined* says that a conforming program should not do it because no definition is provided for what it does (and implicitly it would care what the outcome was if it tried it). It is important to remember, however, that if the syntax permits the statement at all, it must have some outcome in a real implementation.

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

This also implies that an implementation, or another standard, could specify or define the result in a useful fashion. The terms apply to POSIX.1 specifically.

The term *implementation defined* implies requirements for documentation that are not required for *undefined* (or *unspecified*). Where there is no need for a conforming program to know the definition, the term *undefined* is used, even though *implementation defined* could also have been used in this context. There could be a fourth term, specifying "POSIX.1 does not say what this does; it is acceptable to define it in an implementation, but it does not need to be documented," and undefined would then be used very rarely for the few things for which any definition is not useful.

In many places POSIX.1 is silent about the behavior of some possible construct. For example, a variable may be defined for a specified range of values and behaviors are described for those values; nothing is said about what happens if the variable has any other value. That kind of silence can imply an error in the standard, but it may also imply that the standard was intentionally silent and that any behavior is permitted. There is a natural tendency to infer that if the standard is silent, a behavior is prohibited. That is not the intent. Silence is intended to be equivalent to the term *unspecified*.

B.2.2.2 General Terms

Many of these definitions are necessarily circular, and some of the terms (such as *process*) are variants of basic computing science terms that are inherently hard to define. Some are defined by context in the prose topic descriptions of the general concepts in 2.3, but most appear in the alphabetical glossary format of the terms in 2.2.2.

Some definitions must allow extension to cover terms or facilities that are not explicitly mentioned in POSIX.1. For example, the definition of *file* must permit interpretation to include streams, as found in the Eighth Edition (a research version of the UNIX system). The use of abstract intermediate terms (such as *object* in place of, or in addition to, *file*) has mostly been avoided in favor of careful definition of more traditional terms.

Some terms in the following list of notes do not appear in POSIX.1; these are marked prefixed with a asterisk (*). Many of them have been specifically excluded from POSIX.1 because they concern system administration, implementation, or other issues that are not specific to the programming interface. Those are marked with a reason, such as “implementation defined.”

appropriate privileges: One of the fundamental security problems with many historical UNIX systems has been that the privilege mechanism is monolithic—a user has either no privileges or *all* privileges. Thus, a successful “trojan horse” attack on a privileged process defeats all security provisions. Therefore, POSIX.1 allows more granular privilege mechanisms to be defined. For many historical implementations of the UNIX system, the presence of the term *appropriate privileges* in POSIX.1 may be understood as a synonym for *super-user* (UID 0). However, future systems will undoubtedly emerge where this is not the case and each discrete controllable action will have *appropriate privileges* associated with it. Because this mechanism is *implementation defined*, it must be described in the conformance document. Although that description affects several parts of POSIX.1 where the term *appropriate privilege* is used, because the term *implementation defined* only appears here, the description of the entire mechanism and its effects on these other sections belongs in clause 2.3 of the conformance document. This is especially convenient for implementations with a single mechanism that applies in all areas, since it only needs to be described once.

clock tick: The C Standard {2} defines a similar interval for use by the *clock*() function. There is no requirement that these intervals be the same. In historical implementations these intervals are different. Currently only the *times*() function uses values stated in terms of clock ticks, although other functions might use them in the future.

controlling terminal: The question of which of possibly several special files referring to the terminal is meant is not addressed in POSIX.1.

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

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

directory: The format of the directory file is implementation defined and differs radically between System V and 4.3BSD. However, routines (derived from 4.3BSD) for accessing directories are provided in 5.1.2 and certain constraints on the format of the information returned by those routines are made in 5.1.1.

directory entry: Throughout the document, the term *link* is used [about the *link*() function, for example] in describing the objects that point to files from directories.

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

dot-dot: Historical implementations permit the use of these filenames without their special meanings. Such use precludes any meaningful use of these filenames by a Conforming POSIX.1 Application. Therefore, such use is considered an extension, the use of which makes an implementation nonconforming. See also B.2.3.6.

Epoch: Historically, the origin of UNIX system time was referred to as “00:00:00 GMT, January 1, 1970.” Greenwich Mean Time is actually not a term acknowledged by the international standards community; therefore, this term, *Epoch*, is used to abbreviate the reference to the actual standard, Coordinated Universal Time. The concept of leap seconds is added for precision; at the time POSIX.1 was published, 14 leap seconds had been added since January 1, 1970. These 14 seconds are ignored to provide an easy and compatible method of computing time differences.

Most systems' notion of “time” is that of a continuously increasing value, so this value should increase even during leap seconds. However, not only do most systems not keep track of leap seconds, but most systems are probably not synchronized to any standard time reference. Therefore, it is inappropriate to require that a time represented as seconds since the Epoch precisely represent the number of seconds between the referenced time and the Epoch.

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

It is important that the interpretation of time names and *seconds since the Epoch* values be consistent across conforming systems. That is, it is important that all conforming systems interpret “536457599 seconds since the Epoch” as 59 seconds, 59 minutes, 23 hours 31 December 1986, regardless of the accuracy of the system's idea of the current time. The expression is given to assure a consistent interpretation, not to attempt to specify the calendar. The relationship between *tm_yday* and the day of week, day of month, and month is presumed to be specified elsewhere and is not given in POSIX.1.

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

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

Note also that in Section 8, the meaning of *gmtime*(), *localtime*(), and *mktime*() is specified in terms of this expression. However, the C Standard {2} computes *tm_yday* from *tm_mday*, *tm_mon*, and *tm_year* in *mktime*(). Because it is stated as a (bidirectional) relationship, not a function, and because the conversion between month-day-year and day-of-year dates is presumed well known and is also a relationship, this is not a problem.

Note that the expression given will fail after the year 2099. Since the issue of *time_t* overflowing a 32-bit integer occurs well before that time, both of these will have to be addressed in revisions to POSIX.1.

FIFO special file: See *pipe* in B.2.2.2.

file: It is permissible for an implementation-defined file type to be nonreadable or nonwritable.

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

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

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

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

***group file:** Implementation defined; see B.9.

***historical implementations:** This refers to previously existing implementations of programming interfaces and operating systems that are related to the interface specified by POSIX.1. See also “Minimal Changes to Historical Implementations” in the Introduction.

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

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

The term *specific implementation* is sometimes used as a synonym for *implementation*. This should not be interpreted too narrowly; both terms can represent a relatively broad group of systems. For example, a hardware vendor could market a very wide selection of systems that all used the same instruction set, with some systems desktop models and others large multiuser minicomputers. This wide range would probably share a common POSIX.1 operating system, allowing an application compiled for one to be used on any of the others; this is a *[specific] implementation*.

However, that wide range of machines probably has some differences between the models. Some may have different clock rates, different file systems, different resource limits, different network connections, etc., depending on their sizes or intended usages. Even on two identical machines, the system administrators may configure them differently. Each of these different systems is known by the term *a specific instance of a specific implementation*. This term is only used in the portions of POSIX.1 dealing with run-time queries: *sysconf*() and *pathconf*().

***incomplete pathname:** Absolute pathname has been adequately defined.

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

While the job-control facilities supplied by POSIX.1 can, in theory, support different types of interactive job-control interfaces supplied by different types of shells, there is historically one particular interface that is most common (provided by BSD C Shell). This discussion describes that interface as a means of illustrating how the POSIX.1 job-control facilities can be used.

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

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

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

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

Implementing Job Control Shells

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

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

The concept of *process groups* is used to provide this grouping. The shell places each job in a separate process group via the *setpgid*() function. To do this, the *setpgid*() function is invoked by the shell for each process in the job. It is actually useful to invoke *setpgid*() twice for each process: once in the child process, after calling *fork*() to create the process, but before calling one of the *exec* functions to begin execution of the program, and once in the parent shell process, after calling *fork*() to create the child. The redundant invocation avoids a race condition by ensuring that the child process is placed into the new process group before either the parent or the child relies on this being the case. The *process group ID* for the job is selected by the shell to be equal to the *process ID* of one of the processes in the job. Some shells choose to make one process in the job be the parent of the other processes in the job (if any). Other shells (e.g., the C Shell) choose to make themselves the parent of all processes in the pipeline (job). In order to support this latter case, the *setpgid*() function accepts a process group ID parameter since the correct process group ID cannot be inherited from the shell. The shell itself is considered to be a job and is the sole process in its own process group.

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

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

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

There is additional flexibility in the POSIX.1 job-control facilities that allows deviations from the typical interface. Clearing the TOSTOP terminal flag (see 7.1.2.5) allows background jobs to perform *write*() functions without stopping. The same effect can be achieved on a per-process basis by having a process set the signal action for SIGTTOU to SIG_IGN.

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

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

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

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

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

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

Implementing Job Control Applications

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

An application can inadvertently subvert job-control processing by “blindly” altering the handling of signals. A common application error is to learn how many signals the system supports and to ignore or catch them all. Such an application makes the assumption that it does not know what this signal is, but knows the right handling action for it. The system may initialize the handling of job-control stop signals so that they are being ignored. This allows shells that do not support job control to inherit and propagate these settings and hence to be immune to stop signals. A job-control shell will set the handling to the default action and propagate this, allowing processes to stop. In doing so, the job-control shell is taking responsibility for restarting the stopped applications. If an application wishes to catch the stop signals itself, it should first determine their inherited handling states. If a stop signal is being ignored, the application should continue to ignore it. This is directly analogous to the recommended handling of SIGINT described in the UNIX Programmer's Manual {B43} .

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

Implementing Job Control Systems

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

Discussions of the changes can be found in the clauses that discuss the specific interfaces. See B.3.2.1, B.3.2.2, B.3.3.1.1, B.3.3.2, B.3.3.4, B.4.3.1, B.4.3.3, B.7.1.1.4, and B.7.2.4.

It is only useful for a process to be affected by job-control signals if it is the descendant of a job-control shell. Otherwise, there will be nothing that continues the stopped process. Because a job-control shell is allowed, but not required, by POSIX.1, an implementation must provide a mechanism that shields processes from job-control signals when there is no job-control shell. The usual method is for the system initialization process (typically called init), which is the ancestor of all processes, to launch its children with the signal handling action set to SIG_IGN for the signals SIGTSTP, SIGTTIN, and SIGTTOU. Thus, all login shells start with these signals ignored. If the shell is not job-control cognizant, then it should not alter this setting and all its descendants should inherit the same ignored settings. At the point where a job-control shell is launched, it resets the signal handling action for these signals to be SIG_DFL for its children and (by inheritance) their descendants. Also, shells that are not job-control cognizant will not alter the process group of their descendants or of their controlling terminal; this has the effect of making all processes be in the foreground (assuming the shell is in the foreground). While this approach is valid, POSIX.1 added the concept of orphaned process groups to provide a more robust solution to this problem.

All processes in a session managed by a shell that is not job-control cognizant are in an orphaned process group and are protected from stopping.

POSIX.1 does not specify how controlling terminal access is affected by a user logging out (that is, by a controlling process terminating). 4.2BSD uses the *vhangup*() function to prevent any access to the controlling terminal through file descriptors opened prior to logout. System V does not prevent controlling terminal access through file descriptors opened prior to logout (except for the case of the special file, `/dev/tty`). Some implementations choose to make processes immune from job control after logout (that is, such processes are always treated as if in the foreground); other implementations continue to enforce foreground/background checks after logout. Therefore, a Conforming POSIX.1 Application should not attempt to access the controlling terminal after logout since such access is unreliable. If an implementation chooses to deny access to a controlling terminal after its controlling process exits, POSIX.1 requires a certain type of behavior (see 7.1.1.3).

***kernel:** See *system call*.

***library routine:** See *system call*.

***logical device:** Implementation defined.

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

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

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

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

***mounted file system:** See *file system.*

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

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

The following alternate names were discussed:

For *open file description*:

open instance, *file access description*, *open file information*, and *file access information*.

For *file descriptor*:

file handle, file number [c.f., *fileno*()]. Some historical implementations use the term *file table entry*.

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

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

page: The term page is defined to support the description of the behavior of memory mapping for shared memory and memory mapped files, and the description of the behavior of process memory locking. It is not intended to imply that shared memory/file mapping and memory locking are applicable only to "paged" architectures. For the purposes of this part of ISO/IEC 9945, whatever the granularity on which an architecture supports mapping or locking is considered to be a "page." If an architecture cannot support the memory mapping or locking functions specified by the standard on any granularity, then these options will not be implemented on the architecture.

***passwd file:** Implementation defined; see B.9.

parent directory: There may be more than one directory entry pointing to a given directory in some implementations. The wording here identifies that exactly one of those is the parent directory. In 2.3.6, dot-dot is identified as the way that the unique directory is identified. (That is, the parent directory is the one to which dot-dot points.) In the case of a remote file system, if the same file system is mounted several times, it would appear as if they were distinct file systems (with interesting synchronization properties).

pipe: It proved convenient to define a pipe as a special case of a FIFO even though historically the latter was not introduced until System III and does not exist at all in 4.3BSD.

portable filename character set: The encoding of this character set is not specified—specifically, ASCII is not required. But the implementation must provide a unique character code for each of the printable graphics specified by POSIX.1. See also B.2.3.4.

Situations where characters beyond the portable filename character set (or historically ASCII or ISO/IEC 646 {1}) would be used (in a context where the portable filename character set or ISO/IEC 646 {1} is required by POSIX.1) are expected to be common. Although such a situation renders the use technically noncompliant, mutual agreement among the users of an extended character set will make such use portable between those users. Such a mutual agreement could be formalized as an optional extension to POSIX.1. (Making it required would eliminate too many possible systems, as even those systems using ISO/IEC 646 {1} as a base character set extend their character sets for Western Europe and the rest of the world in different ways.)

Nothing in POSIX.1 is intended to preclude the use of extended characters where interchange is not required or where mutual agreement is obtained. It has been suggested that in several places "should" be used instead of "shall." Because (in the worst case) use of any character beyond the portable filename character set would render the program or data not portable to all possible systems, no extensions are permitted in this context.

regular file: POSIX.1 does not intend to preclude the addition of structuring data (e.g., record lengths) in the file, as long as such data is not visible to an application that uses the features described in POSIX.1.

root directory: This definition permits the operation of *chroot*(), even though that function is not in POSIX.1. See also *file hierarchy*.

***root file system:** Implementation defined.

***root of a file system:** Implementation defined. See *mount point*.

seconds since the Epoch: The formula here is not precisely correct for leap centuries. See the discussion for *Epoch* for further details.

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

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

synchronously generated signal: Those signals that may be generated synchronously include SIGABRT, SIGBUS, SIGILL, SIGFPE, SIGPIPE, SIGSEGV.

***system call:** The distinction between a *system call* and a *library routine* is an implementation detail that may differ between implementations and has thus been excluded from POSIX.1. See “Interface, Not Implementation” in the Introduction.

system reboot: A system reboot is an event initiated by an unspecified circumstance that causes all processes (other than special system processes) to be terminated in an implementation-defined manner, after which any changes to the state and contents of files created or written to by a Conforming POSIX.1 Application prior to the event are implementation-defined.

synchronized I/O data integrity completion-synchronized I/O file integrity completion: These terms specify that for synchronized read operations, pending writes must be successfully completed before the read operation can complete. This is motivated by two circumstances. Firstly, when synchronizing processes can access the same file, but not share common buffers (such as for a remote file system), this requirement permits the reading process to guarantee that it can read data written remotely. Secondly, having data written synchronously is insufficient to guarantee the order with respect to a subsequent write by a reading process, and thus this extra read semantic is necessary.

thread: This part of ISO/IEC 9945 defines a thread to be a flow of control within a process. Each thread has a minimal amount of private state; most of the state associated with a process is shared among all of the threads in the process. While most multithread extensions to POSIX have taken this approach, others have made different decisions.

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

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

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

thread ID: Separate programs should communicate through well-defined interfaces and should not depend on each other's implementation. For example, if a programmer decides to rewrite the sort program using multiple threads, it should be easy to do this so that the interface to the sort program does not change. Consider that if the user causes SIGINT to be generated while the sort program is running, keeping the same interface means that the entire sort program is killed, not just one of its threads. As another example, consider a real-time program that manages a reactor. Such a program may wish to allow other programs to control the priority at which it watches the control rods. One technique to accomplish this is to write the ID of the thread watching the control rods into a file and allow other programs to change the priority of that thread as they see fit. A simpler technique is to have the reactor process accept IPCs (Inter-Process Communication messages) from other processes, telling it at a semantic level what priority the program should assign to watching the control rods. This allows the programmer greater flexibility in the implementation. For example, the programmer can change the implementation from having one thread per rod to having one thread watching all of the rods without changing the interface. Having threads live inside the process means that the implementation of a process is invisible to outside processes (excepting debuggers and system management tools).

Threads do not provide a protection boundary. Every thread model allows threads to share memory with other threads and encourages this sharing to be widespread. This means that one thread can wipe out memory that is needed for the correct functioning of other threads that are sharing its memory. Consequently, providing each thread with its own user and/or group IDs would not provide a protection boundary between threads sharing memory.

thread-safe function: All functions required by this part of ISO/IEC 9945 need to be thread-safe. Implementations have to provide internal synchronization when necessary in order to achieve this goal. In certain cases, e.g., most floating-point implementations, context switch code may have to manage the writable shared state.

See the discussion under thread-safety.

It is not required that all functions provided by this part of ISO/IEC 9945 be either async-cancel safe or async-signal safe.

Thread-Safety and Locking of Existing Functions

Originally, POSIX.1 was not designed to work in a multithreaded environment, and some implementations of some existing functions will not work properly when executed concurrently. To provide routines that will work correctly in an environment with threads ("thread-safe"), two problems need to be solved:

1) Routines that maintain or return pointers to static areas internal to the routine (which may now be shared) need to be modified. The routines *ttyname*(), 4.7.2, and *localtime* (), 8.1, are examples.
2) Routines that access data space shared by more than one thread need to be modified. The *malloc*(), 8.1, and *stdio*, 8.2, routines are examples.

There are a variety of constraints on these changes. The first is compatibility with the existing versions of these functions—non-thread-safe functions will continue to be in use for some time, as the original interfaces are used by existing code. Another is that the new thread-safe versions of these functions represent as small a change as possible over the familiar interfaces provided by the existing non-thread-safe versions. The new interfaces should be independent of any particular threads implementation. In particular, they should be thread-safe without depending on explicit thread-specific memory. Finally, there should be minimal performance penalty due to the changes made to the functions.

It is intended that the list of functions from POSIX.1 that cannot be made threadsafe and for which corrected versions are provided be complete.

Thread-Safety and Locking Solutions

Many of the POSIX.1 functions were thread-safe and did not change at all. However, some functions (for example, the math functions typically found in *libm*) are not thread-safe because of writable shared global state. For instance, in IEEE Std 754-1985 standard floating-point implementations, the computation modes and flags are global and shared.

Some functions are not thread-safe because a particular implementation is not reentrant, typically because of a nonessential use of static storage. These require only a new implementation.

Thread-safe libraries are useful in a wide range of parallel (and asynchronous) programming environments, not just within pthreads. In order to be used outside the context of pthreads, however, such libraries still have to use some synchronization method. These could either be independent of the pthread synchronization operations, or they could be a subset of the pthread interfaces. Either method results in thread-safe library implementations that can be used without the rest of pthreads.

Some functions, such as the *stdio* interface and dynamic memory allocation functions such as *malloc*(), are interdependent routines that share resources (for example, buffers) across related calls. These require synchronization to work correctly, but they do not require any change to their external (user-visible) interfaces.

In some cases, such as *getc*() and *putc*(), adding synchronization is likely to create an unacceptable performance impact. In this case, slower thread-safe synchronized functions are to be provided, but the original, faster (but unsafe) functions (which may be implemented as macros) are retained under new names. Some additional special-purpose synchronization facilities are necessary for these macros to be usable in multithreaded programs. This also requires changes in `<stdio.h>`.

The other common reason that functions are unsafe is that they return a pointer to static storage, making the functions non-thread-safe. This has to be changed, and there are three natural choices:

1) *Return a pointer to thread-specific storage*. This could incur a severe performance penalty on those architectures with a costly implementation of the thread-specific data interface.
   A variation on this technique is to use *malloc*() to allocate storage for the function output and return a pointer to this storage. This technique may also have an undesirable performance impact, however, and a simplistic implementation requires that the user program explicitly free the storage object when it is no longer needed. This technique is used by some existing POSIX.1 functions. With careful implementation for infrequently used functions, there may be little or no performance or storage penalty, and the maintenance of already-standardized interfaces is a significant benefit.
2) *Return the actual value computed by the function.* This technique can only be used with functions that return pointers to structures—routines that return character strings would have to wrap their output in an enclosing structure in order to return the output on the stack. There is also a negative performance impact inherent in this solution in that the output value has to be copied twice before it can be used by the calling function: once from the called routine's local buffers to the top of the stack, then from the top of the stack to the assignment target. Finally, many older compilers cannot support this technique due to a historical tendency to use internal static buffers to deliver the results of structure-valued functions.
3) *Have the caller pass the address of a buffer to contain the computed value.* The only disadvantage of this approach is that extra arguments have to be provided by the calling program. It represents the most efficient solution to the problem, however, and, unlike the *malloc*() technique, it is semantically clear.

There are some routines (often groups of related routines) whose interfaces are inherently non-thread-safe because they communicate across multiple function invocations by means of static memory locations. The solution is to redesign the calls so that they are thread-safe, typically by passing the needed data as extra parameters. Unfortunately, this may require major changes to the interface as well.

A floating-point implementation using IEEE Std 754-1985 is a case in point. A less problematic example is the *rand48* family of pseudorandom number generators. The functions *getgrgid*(), *getgrnam*(), *getpwnam*(), and *getpwuid*() are another such case.

The problems with *errno* are discussed in B.2.4.

Asynchronous Safety and Thread-Safety

A floating-point implementation has many modes that effect rounding and other aspects of computation. Functions in some math library implementations may change the computation modes for the duration of a function call. If such a function call is interrupted by a signal or cancellation, the floating-point state is not required to be protected.

There is a significant cost to make floating-point operations async-cancel safe or async-signal safe; accordingly, neither form of async safety is required.

Functions Returning Pointers to Static Storage

For those functions that are not thread-safe because they return values in fixed size statically allocated structures, alternate “_r” forms are provided that pass a pointer to an explicit result structure. Those that return pointers into library-allocated buffers have forms provided with explicit buffer and length parameters.

For functions that return pointers to library-allocated buffers, it makes sense to provide “_r” versions that allow the application control over allocation of the storage in which results are returned. This allows the state used by these functions to be managed on an application-specific basis, supporting per-thread, per-process, or other application-specific sharing relationships.

Previous drafts of this part of ISO/IEC 9945 had provided "_r" versions for functions that returned pointers to variable-size buffers without providing a means for determining the required buffer size. This would have made using such functions exceedingly clumsy, potentially requiring iteratively calling them with increasingly larger guesses for the amount of storage required. Hence, *sysconf*() variables have been provided for such functions that return the maximum required buffer size.

Thus, the rule that has been followed by this part of ISO/IEC 9945 when adapting single-threaded non-thread-safe library functions is as follows: all functions returning pointers to library-allocated storage should have "_r" versions provided, allowing the application control over the storage allocation. Those with variable-sized return values accept both a buffer address and a length parameter. The *sysconf*() variables are provided to supply the appropriate buffer sizes when required. Implementors are encouraged to apply the same rule when adapting their own existing functions to a pthreads environment.

***virtual processor:** The term "virtual processor" was chosen as a neutral term describing all kernel-level scheduleable entities, such as processes, Mach tasks, or lightweight processes. Implementing threads using multiple processes as virtual processors, or implementing multiplexed threads above a virtual processor layer, should be possible, provided some mechanism has also been implemented for sharing state between processes or virtual processors. Many systems may also wish to provide implementations of threads on systems providing "shared processes" or "variable-weight processes." It was felt that exposing such implementation details would severely limit the type of systems upon which the threads interface could be supported and prevent certain types of valid implementations. It was also determined that a virtual processor interface was out of the scope of this part of ISO/IEC 9945.

B.2.2.3 Abbreviations

POSIX.0: Although this term is not used in the normative text of this part of ISO/IEC 9945, it is used in this rationale to refer to IEEE Std 1003.0-1995 .

POSIX.1b: Although this term is not used in the normative text for this part of ISO/IEC 9945, it is used in this rationale to refer to the elements of the realtime extension amendment. (This was earlier referred to as POSIX.4 during the standard development process.)

POSIX.1c: Although this term is not used in the normative text of this part of ISO/IEC 9945, it is used in this rationale to refer to the threads extension amendment. (This was earlier referred to as POSIX.4a during the standard development process.)

B.2.3 General Concepts

B.2.3.1 extended security controls: Allowing an implementation to define extended security controls enables the use of POSIX.1 in environments that require different or more rigorous security than that provided in POSIX.1. Extensions are allowed in two areas: privilege and file access permissions. The semantics of these areas have been defined to permit extensions with reasonable, but not exact, compatibility with all existing practices. For example, the elimination of the super-user definition precludes identifying a process as privileged or not by virtue of its effective user ID.

B.2.3.2 file access permissions: A process should not try to anticipate the result of an attempt to access data by *a priori* use of these rules. Rather, it should make the attempt to access data and examine the return value (and possibly *errno* as well), or use *access*(). An implementation may include other security mechanisms in addition to those specified in POSIX.1, and an access attempt may fail because of those additional mechanisms, even though it would succeed according to the rules given in this subclause. (For example, the user's security level might be lower than that of the object of the access attempt.) The optional supplementary group IDs provide another reason for a process to not attempt to anticipate the result of an access attempt.

B.2.3.3 file hierarchy: Though the file hierarchy is commonly regarded to be a tree, POSIX.1 does not define it as such for three reasons:

1) Links may join branches.
2) In some network implementations, there may be no single absolute root directory. See *pathname resolution*.
3) With symbolic links (found in 4.3BSD), the file system need not be a tree or even a directed acyclic graph.

B.2.3.4 file permissions: Examples of implementation-defined constraints that may deny access are mandatory labels and access control lists. Historically, certain filenames have been reserved. This list includes `core`, `/etc/passwd`, etc. Portable applications should avoid these.

Most historical implementations prohibit case folding in filenames; i.e., treating upper- and lowercase alphabetic characters as identical. However, some consider case folding desirable:

— For user convenience
— For ease of implementation of the POSIX.1 interface as a hosted system on some popular operating systems, which is compatible with the goal of making the POSIX.1 interface broadly implementable (see "Broadly Implementable" in the Introduction)

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

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

1) No solid evidence has been produced as to whether case sensitivity or case insensitivity is more convenient for users.
2) Making case insensitivity a POSIX.1 implementation option would be worse than either having it or not having it, because
   a) More confusion would be caused among users.
   b) Application developers would have to account for both cases in their code.
   c) POSIX.1 implementors would still have other problems with native file systems, such as short or otherwise constrained filenames or pathnames, and the lack of hierarchical directory structure.
3) Case folding is not easily defined in many European languages, both because many of them use characters outside the USASCII alphabetic set, and because
   a) In Spanish, the digraph `ll` is considered to be a single letter, the capitalized form of which may be either `Ll` or `LL`, depending on context.
   b) In French, the capitalized form of a letter with an accent may or may not retain the accent depending on the country in which it is written.
   c) In German, the sharp ess may be represented as a single character resembling a Greek beta (β) in lowercase, but as the digraph SS in uppercase.
   d) In Greek, there are several lowercase forms of some letters; the one to use depends on its position in the word. Arabic has similar rules.
4) Many East Asian languages, including Japanese, Chinese, and Korean, do not distinguish case and are sometimes encoded in character sets that use more than one byte per character.
5) Multiple character codes may be used on the same machine simultaneously. There are several ISO character sets for European alphabets. In Japan, several Japanese character codes are commonly used together, sometimes even in filenames; this is evidently also the case in China. To handle case insensitivity, the kernel would have to at least be able to distinguish for which character sets the concept made sense.
6) The file system implementation historically deals only with bytes, not with characters, except for slash and the null byte.
7) The purpose of POSIX.1 is to standardize the common, existing definition (see "Application Oriented" in the Introduction) of the UNIX system programming interface, not to change it. Mandating case insensitivity would make all historical implementations nonstandard.
8) Not only the interface, but also application programs would need to change, counter to the purpose of having minimal changes to existing application code.
9) At least one of the original developers of the UNIX system has expressed objection in the strongest terms to either requiring case insensitivity or making it an option, mostly on the basis that POSIX.1 should not hinder portability of application programs across related implementations in order to allow compatibility with unrelated operating systems.

Two proposals were entertained regarding case folding in filenames:

— Remove all wording that previously permitted case folding.
  Rationale: Case folding is inconsistent with portable filename character set definition and filename definition (all characters except slash and null). No known implementations allowing all characters except slash and null also do case folding.
— Change "though this practice is not recommended:" to "although this practice is strongly discouraged."
  Rationale: If case folding must be included in POSIX.1, the wording should be stronger to discourage the practice.

The consensus selected the first proposal. Otherwise, a portable application would have to assume that case folding would occur when it was not wanted, but that it would not occur when it was wanted.

B.2.3.5 file times update: This subclause reflects the actions of historical implementations. The times are not updated immediately, but are only marked for update by the functions. An implementation may update these times immediately.

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

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

B.2.3.6 pathname resolution: What the filename dot-dot refers to relative to the root directory is implementation defined. In Version 7 it refers to the root directory itself; this is the behavior mentioned in the standard. In some networked systems the construction `/../hostname/` is used to refer to the root directory of another host, and POSIX.1 permits this behavior.

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

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

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

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

B.2.3.7 concurrent execution: There is no additional rationale provided for this subclause.

B.2.3.8 memory synchronization: In older multiprocessors, access to memory by the processors was strictly multiplexed. This meant that a processor executing program code interrogates or modifies memory in the order specified by the code and that all the memory operation of all the processors in the system appear to happen in some global order, though the operation histories of different processors are interleaved arbitrarily. The memory operations of such machines are said to be sequentially consistent. In this environment, threads can synchronize using ordinary memory operations. For example, a producer thread and a consumer thread can synchronize access to a circular data buffer as follows:

```
        int rdptr = 0;
        int wrptr = 0;
        data_t buf[BUFSIZE];
Thread 1:
        while (work_to_do) {
                int next;
                buf[wrptr] = produce();
                next = (wrptr + 1) % BUFSIZE;
                while (rdptr == next)
                        ;
                wrptr = next;
        }
Thread 2:
        while (work_to_do) {
                while (rdptr == wrptr)
                        ;
                consume (buf[rdptr]);
                rdptr = (rdptr + 1) % BUFSIZE;
        }
```

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

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

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

Conforming applications may only use the functions listed to synchronize threads of control with respect to memory access. There are many other candidates for functions that might also be used. Examples are: signal sending and reception, or pipe writing and reading. In general, any function that allows one thread of control to wait for an action caused by another thread of control is a candidate. This part of ISO/IEC 9945 does not require these additional functions to synchronize memory access since this would imply the following:

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

Formal definitions of the memory model were rejected as unreadable by the vast majority of programmers. In addition, most of the formal work in the literature has concentrated on the memory as provided by the hardware as opposed to the application programmer through the compiler and runtime system. It was believed that a simple statement intuitive to most programmers would be most effective. This part of ISO/IEC 9945 defines functions that can be used to synchronize access to memory, but it leaves open exactly how one relates those functions to the semantics of each function as specified elsewhere in this part of ISO/IEC 9945. This part of ISO/IEC 9945 also does not make a formal specification of the partial ordering in time that the functions can impose, as that is implied in the description of the semantics of each function. It simply states that the programmer has to ensure that modifications do not occur "simultaneously" with other access to a memory location.

B.2.3.9 thread-safety: Where the interface of a function required by POSIX.1 or the C Standard {2} precludes thread-safety, an alternate form that shall be thread-safe is provided. The names of these thread-safe forms are the same as the non-thread-safe forms with the addition of the suffix "_r." The suffix "_r" is historical, where the "r" stood for "reentrant." The term "reentrant" has since been replaced by the term "thread-safe."

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

B.2.4 Error Numbers

The C Standard {2} requires that *errno* be an assignable *lvalue*. Originally, the definition in POSIX.1 was stricter than that in the C Standard {2}, *extern int errno*, in order to support historical usage. In a multithreaded environment, implementing *errno* as a global variable results in non-deterministic results when accessed. It is required, however, that *errno* work as a per-thread error reporting mechanism. In order to do this, a separate *errno* value has to be maintained for each thread. The following subclause discusses the various alternative solutions that were considered.

In order to avoid this problem altogether for new functions, these functions avoid using *errno* and, instead, return the error number directly as the function return value; a return value of zero indicates that no error was detected.

For any function that can return errors, the function return value is not used for any purpose other than for reporting errors. Even when the output of the function is scalar, it is passed through a function argument. While it might have been possible to allow some scalar outputs to be coded as negative function return values and mixed in with positive error status returns, this was rejected—using the return value for a mixed purpose was judged to be of limited use and error prone.

Checking the value of *errno* alone is not sufficient to determine the existence or type of an error, since it is not required that a successful function call clear *errno*. The variable *errno* should only be examined when the return value of a function indicates that the value of *errno* is meaningful. In that case, the function is required to set the variable to something other than zero.

A successful function call may set the value of *errno* to zero, or to any other value (except where specifically prohibited; see B.5.4.1). But it is meaningless to do so, since the value of *errno* is undefined except when the description of a function explicitly states that it is set, and no function description states that it should be set on a successful call. Most functions in most implementations do not change *errno* on successful completion. Exceptions are *isatty*() and *ptrace*(). The latter is not in POSIX.1, but is widely implemented and clears *errno* when called. The value of *errno* is not defined unless all signal handlers that use functions that could change *errno* save and restore it.

POSIX.1 requires (in the Errors subclauses of function descriptions) certain error values to be set in certain conditions because many existing applications depend on them. Some error numbers, such as [EFAULT], are entirely implementation defined and are noted as such in their description in 2.4. This subclause otherwise allows wide latitude to the implementation in handling error reporting.

Some of the Errors clauses in POSIX.1 have two subclauses. The first:

> "If any of the following conditions occur, the *foo*() function shall return –1 and set *errno* to the corresponding value:"

could be called the "mandatory" subclause. The second:

> "For each of the following conditions, when the condition is detected, the *foo*() function shall return −1 and set *errno* to the corresponding value:"

could be informally known as the "optional" subclause. This latter subclause has evolved in meaning over time. In early drafts, it was only used for error conditions that could not be detected by certain hardware configurations, such as the [EFAULT] error, as described below. The subclause recently has also added conditions associated with optional system behavior, such as job control errors. Attempting to infer the quality of an implementation based on whether it detects such conditions is not useful.

Following each one-word symbolic name for an error, there is a one-line tag, which is followed by a description of the error. The one-line tag is merely a mnemonic or historical referent and is not part of the specification of the error. Many programs print these tags on the standard error stream [often by using the C Standard {2} *perror*() function] when the corresponding errors are detected, but POSIX.1 does not require this action.

[ECANCELED]	
	This spelling was chosen as being more common.
[EFAULT]	Most historical implementations do not catch an error and set *errno* when an invalid address is given to the functions *wait*(), *time*(), or *times*(). Some implementations cannot reliably detect an invalid address. And most systems that detect invalid addresses will do so only for a system call, not for a library routine.
[EFTYPE]	This error code was proposed in earlier drafts as "Inappropriate operation for file type," meaning that the operation requested is not appropriate for the file specified in the function call. This code was proposed, although the same idea was covered by [ENOTTY], because the connotations of the name would be misleading. It was pointed out that the *fcntl*() function uses the error code [EINVAL] for this notion, and hence all instances of [EFTYPE] were changed to this code.
[EINTR]	POSIX.1 prohibits conforming implementations from restarting interrupted system calls. However, it does not require that [EINTR] be returned when another legitimate value may be substituted; e.g., a partial transfer count when *read*() or *write*() are interrupted. This is only given when the signal catching function returns normally as opposed to returns by mechanisms like *longjmp*() or *siglongimp*().
[ENOMEM]	The term *main memory* is not used in POSIX.1 because it is implementation defined.
[ENOTSUP]	This error code is to be used when an implementation chooses to implement the required functionality of this standard but does not support optional facilities defined by this standard. The return of [ENOSYS] is to be taken to indicate that the function of the interface is not supported at all; the function will always fail with this error code.
[ENOTTY]	The symbolic name for this error is derived from a time when device control was done by *ioctl*() and that operation was only permitted on a terminal interface. The term "TTY" is derived from *teletypewriter*, the devices to which this error originally applied.
[EPIPE]	This condition normally generates the signal SIGPIPE; the error is returned if the signal does not terminate the process.
[EROFS]	In historical implementations, attempting to *unlink*() or *rmdir*() a mount point would generate an [EBUSY] error. An implementation could be envisioned where such an operation could be performed without error. In this case, if *either* the directory entry or the actual data structures reside on a read-only file system, [EROFS] is the appropriate error to generate. (For example, changing the link count of a file on a read-only file system could not be done, as is required by *unlink*(), and thus an error should be reported.)

Two error numbers, [EDOM] and [ERANGE], were added to this subclause primarily for consistency with the C Standard {2}.

B.2.4.1 Alternative Solutions for Per-Thread *errno*

The usual implementation of *errno* as a single global variable does not work in a multithreaded environment. In such an environment, a thread may make a POSIX.1 call and get a –1 error return, but before that thread can check the value of *errno*, another thread might have made a second POSIX.1 call that also set *errno*. This behavior is unacceptable in robust programs. There were a number of alternatives that were considered for handling the *errno* problem:

1) Implement *errno* as a per-thread integer variable.
2) Implement *errno* as a service that can access the per-thread error number.
3) Change all POSIX.1 calls to accept an extra status argument and avoid setting *errno*.
4) Change all POSIX.1 calls to raise a language exception.

The first option offers the highest level of compatibility with existing practice but requires special support in the linker, compiler, and/or virtual memory system to support the new concept of thread private variables. When compared with current practice, the third and fourth options are much cleaner, more efficient, and encourage a more robust programming style, but they require new versions of all of the POSIX.1 functions that might detect an error. The second option offers compatibility with existing code that uses the C Standard {2} language header `<errno.h>` to define the symbol *errno*. In this option, *errno* may be a macro defined:

```
#define errno        (*__errno())
extern int           *__errno();
```

This option may be implemented as a per-thread variable whereby an *errno* field is allocated in the user space object representing a thread, and whereby the function *__errno*() makes a system call to determine the location of its user space object and returns the address of the *errno* field of that object. Another implementation, one that avoids calling the kernel, involves allocating stacks in chunks. The stack allocator keeps a side table indexed by chunk number containing a pointer to the thread object that uses that chunk. The *__errno*() function then looks at the stack pointer, determines the chunk number, and uses that as an index into the chunk table to find its thread object and thus its private value of *errno*. On most architectures, this can be done in four to five instructions. Some compilers may wish to implement *__errno*() inline to improve performance.

B.2.4.2 Disallowing Return of the [EINTR] Error Code

Many blocking interfaces defined by POSIX.1 may return [EINTR] if interrupted during their execution by a signal handler. Blocking interfaces introduced under the {_POSIX_THREADS} option do not have this property. Instead, they require that the interface appear to be atomic with respect to interruption. In particular, clients of block interfaces need not handle any possible [EINTR] return as a special case since it will never occur. If it is necessary to restart operations or complete incomplete operations following the execution of a signal handler, this is handled by the implementation, rather than by the application.

Requiring applications to handle [EINTR] errors on blocking interfaces has been shown to be a frequent source of often unreproducible bugs, and it adds no compelling value to the available functionality. Thus, blocking interfaces introduced for use by multithreaded programs do not use this paradigm. In particular, in none of the functions *flockfile*(), *pthread_cond_timedwait*(), *pthread_cond_wait*(), *pthread_join*(), *pthread_mutex_lock*(), and *sigwait*() did providing [EINTR] returns add value, or even particularly make sense. Thus, these functions do not provide for an [EINTR] return, even when interrupted by a signal handler. The same arguments can be applied to *sem_wait*(), *sem_trywait*(), *sigwaitinfo*(), and *sigtimedwait*(), but implementations are permitted to return [EINTR] error codes for these functions for compatibility with earlier versions of this standard. Applications cannot rely on calls to these functions returning [EINTR] error codes when signals are delivered to the calling thread, but they should allow for the possibility.

B.2.5 Primitive System Data Types

The requirement that additional types defined in this subclause end in "_t" was prompted by the problem of namespace pollution (see B.2.7.2). It is difficult to define a type (where that type is not one defined by POSIX.1) in one header file and use it in another without adding symbols to the namespace of the program. To allow implementors to provide their own types, all POSIX.1 conforming applications are required to avoid symbols ending in "_t", which permits the implementor to provide additional types. Because a major use of types is in the definition of structure members, which can (and in many cases must) be added to the structures defined in POSIX.1, the need for additional types is compelling.

The types such as *ushort* and *ulong*, which are in common usage, are not defined in POSIX.1 (although *ushort_t* would be permitted as an extension). They can be added to `<sys/types.h>` using a feature test macro (see 2.7.2). A suggested symbol for these is _SYSIII. Similarly, the types like *u_short* would probably be best controlled by _BSD.

Some of these symbols may appear in other headers; see 2.7.

dev_t This type may be made large enough to accommodate host-locality considerations of networked systems.

This type must be arithmetic. Earlier drafts allowed this to be nonarithmetic (such as a structure) and provided a *samefile*() function for comparison.

gid_t Some implementations had separated *gid_t* from *uid_t* before POSIX.1 was completed. It would be difficult for them to coalesce them when it was unnecessary. Additionally, it is quite possible that user IDs might be different than group IDs because the user ID might wish to span a heterogeneous network, where the group ID might not.

For current implementations, the cost of having a separate *gid_t* will be only lexical.

mode_t This type was chosen so that implementations could choose the appropriate integral type, and for compatibility with the C Standard {2}. 4.3BSD uses *unsigned short* and the *SVID* uses *ushort*, which is the same. Historically, only the low-order sixteen bits are significant.

nlink_t This type was introduced in place of *short* for *st_nlink* (see 5.6.1) in response to an objection that *short* was too small.

off_t This type is used only in *lseek*(), *fcntl*(), and `<sys/stat.h>`. Many implementations would have difficulties if it were defined as anything other than *long*. Requiring an integral type limits the capabilities of *lseek*() to four gigabytes. See the description of *lread*() in B.6.4. Also, the C Standard {2} supplies routines that use larger types: see *fgetpos*() and *fsetpos*() in B.6.5.3.

pid_t The inclusion of this symbol was controversial because it is tied to the issue of the representation of a process ID as a number. From the point of view of a portable application, process IDs should be "magic cookies"[15] that are produced by calls such as *fork*(), used by calls such as *waitpid*() or *kill*(), and not otherwise analyzed (except that the sign is used as a flag for certain operations).

The concept of a {PID_MAX} value interacted with this in early drafts. Treating process IDs as an opaque type both removes the requirement for {PID_MAX} and allows systems to be more flexible in providing process IDs that span a large range of values, or a small one.

Since the values in *uid_t*, *gid_t*, and *pid_t* will be numbers generally, and potentially both large in magnitude and sparse, applications that are based on arrays of objects of this type are unlikely to be fully portable in any case. Solutions that treat them as magic cookies will be portable.

{CHILD_MAX} precludes the possibility of a "toy implementation," where 'there would only be one process.

[15]An historical term meaning: "An opaque object, or token, of determinate size, whose significance is known only to the entity which created it. An entity receiving such a token from the generating entity may only make such use of the 'cookie' as is defined and permitted by the supplying entity."

ssize_t This is intended to be a signed analog of *size_t*. The wording is such that an implementation may either choose to use a longer type or simply to use the signed version of the type that underlies *size_t*. All functions that return *ssize_t* [*read*() and *write*()] describe as "implementation defined" the result of an input exceeding {SSIZE_MAX}. It is recognized that some implementations might have *ints* that are smaller than *size_t*. A portable application would be constrained not to perform I/O in pieces larger than [SSIZE_MAX), but a portable application using extensions would be able to use the full range if the implementation provided an extended range, while still having a single type-compatible interface.

The symbols *size_t* and *ssize_t* are also required in `<unistd.h>` to minimize the changes needed for calls to *read*() and *write*(). Implementors are reminded that it must be possible to include both `<sys/types.h>` and `<unistd.h>` in the same program (in either order) without error.

uid_t Before the addition of this type, the data types used to represent these values varied throughout early drafts. The `<sys/stat.h>` header defined these values as type *short*, the `<passwd.h>` file (now `<pwd.h>` and `<grp.h>`) used an *int*, and *getuid*() returned an *int*. In response to a strong objection to the inconsistent definitions, all the types to were switched to *uid_t*.

In practice, those historical implementations that use varying types of this sort can typedef *uid_t* to *short* with no serious consequences.

The problem associated with this change concerns object compatibility after structure size changes. Since most implementations will define *uid_t* as a short, the only substantive change will be a reduction in the size of the *passwd* structure. Consequently, implementations with an overriding concern for object compatibility can pad the structure back to its current size. For that reason, this problem was not considered critical enough to warrant the addition of a separate type to POSIX.1.

The types *uid_t* and *gid_t* are magic cookies. There is no {UID_MAX} defined by POSIX.1, and no structure imposed on *uid_t* and *gid_t* other than that they be positive arithmetic types. (In fact, they could be *unsigned char*.) There is no maximum or minimum specified for the number of distinct user or group IDs.

B.2.6 Environment Description

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

LC_* The description of the environment variable names starting with the characters "**LC_**" acknowledges the fact that the interfaces presented in the current version of POSIX.1 are not complete and may be extended as new international functionality is required. In the C Standard {2}, names preceded by "**LC_**" are reserved in the name space for future categories.

To avoid name clashes, new categories and environment variables are divided into two classifications: implementation independent and implementation dependent.

Implementation-independent names will have the following format:

LC_*NAME*

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

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

LC_*name*

PATH Many historical implementations of the Bourne shell do not interpret a trailing colon to represent the current working directory and are thus nonconforming. The C Shell and the KornShell conform to POSIX.1 on this point. The usual name of dot may also be used to refer to the current working directory.

TZ	See 8.1.1 for an explanation of the format.
LOGNAME	4.3BSD uses the environment variable **USER** for this purpose. In most implementations, the value of such a variable is easily forged, so security-critical applications should rely on other means of determining user identity. **LOGNAME** is required to be constructed from the portable filename character set for reasons of interchange. No diagnostic condition is specified for violating this rule, and no requirement for enforcement exists. The intent of the requirement is that if extended characters are used, the “guarantee” of portability implied by a standard is voided. (See also B.2.2.2.)

The following environment variables have been used historically as indicated. However, such use was either so variant as to not be amenable to standardization or to be relevant only to other facilities not specified in POSIX.1, and they have therefore been excluded. They may or may not be included in future POSIX standards. Until then, writers of conforming applications should be aware that details of the use of these variables are likely to vary in different contexts.

IFS	Characters used as field separators.
MAIL	System mailer information.
PS1	Prompting string for interactive programs.
PS2	Prompting string for interactive programs.
SHELL	The shell command interpreter name.

B.2.7 C Language Definitions

The construct `<name.h>` for headers is also taken from the C Standard {2}.

B.2.7.1 Symbols From the C Standard

The reservation of identifiers is paraphrased from the C Standard {2}. The text is included because it needs to be part of POSIX.1, regardless of possible changes in future versions of the C Standard {2}. The reservation of other namespaces is particularly for `<errno.h>`.

These identifiers may be used by implementations, particularly for feature test macros. Implementations should not use feature test macro names that might be reasonably used by a standard.

The requirement for representing the number of clock ticks in 24 h refers to the interval defined by POSIX.1, not to the interval defined by the C Standard {2}.

Including headers more than once is a reasonably common practice, and it should be carried forward from the C Standard {2}. More significantly, having definitions in more than one header is explicitly permitted. Where the potential declaration is “benign” (the same definition twice) the declaration can be repeated, if that is permitted by the compiler. (This is usually true of macros, for example.) In those situations where a repetition is not benign (e.g., typedefs), conditional compilation must be used. The situation actually occurs both within the C Standard {2} and within POSIX.1: *time_t* should be in `<sys/types.h>`, and the C Standard {2} mandates that it be in `<time.h>`. POSIX.1 requires using `<sys/types.h>` with `<time.h>` because of the common-usage environment.

B.2.7.2 POSIX.1 Symbols

This subclause addresses the issue of “namespace pollution.” The C Standard {2} requires that the namespace beyond what it reserves not be altered except by explicit action of the application writer. This subclause defines the actions to add the POSIX.1 symbols for those headers where both the C Standard {2} and POSIX.1 need to define symbols. Where there are nonoverlapping uses of headers, there is no problem.

The list of symbols defined in the C Standard {2} is summarized in the rationale associated with C.

Implementors should note that the requirement on type conversion disallows using an older declaration as a prototype and in effect requires that the number of arguments in the prototype, match that given in POSIX.1.

When headers are used to provide symbols, there is a potential for introducing symbols that the application writer cannot predict. Ideally, each header should only contain one set of symbols, but this is not practical for historical reasons. Thus, the concept of feature test macros is included. This is done in a general manner because it is expected that future additions to POSIX.1 and other related standards will have this same problem. (Future standards not constrained by historical practice should avoid the problem by using new header files rather than using ones already extant.)

This idea is split into two subclauses: 2.7.2.1 covers the case of the C Standard {2} conformant systems, where the requirements of the C Standard {2} are that unless specifically requested the application will not see any other symbols, and "Common Usage," where the default set of symbols is not well controlled and backwards compatibility is an issue.

The common usage case is the more difficult to define. In the C Standard {2} case, each feature test macro simply adds to the possible symbols. In common usage, having _POSIX C SOURCE defined with a value of 199309L is a special case in that it reduces the set to the sum of the C Standard {2} and POSIX.1. (The developers of the C Standard {2} will determine if they want a similar macro to limit the features to just the C Standard {2}; the wording permits this because under those circumstances _POSIX_C_SOURCE would be just another ordinary feature test macro. The only order requirement is "before headers.")

If _POSIX_C_SOURCE is not defined in a common-usage environment, the user presumably gets the same results as in previous releases. Some applications may today be conformant without change, so they would continue to compile as long as common usage is provided. When the C Standard {2} is the default they will have to change (unless they are already C Standard {2} conformant), but this can be done gradually.

Note that the net result of defining _POSIX_C_SOURCE at the beginning of a program is in either case the same: the implementation-defined symbols are only visible if they are requested. (But if _POSIX_C_SOURCE is not used, the implementation default, which is probably backwards compatible, determines their visibility.)

Since_POSIX_SOURCE as specified by IEEE Std 1003.1-1990 did not have a value associated with it, the _POSIX_C_SOURCE macro replaces it, allowing an application to inform the system of the version of the standard to which it conforms. This symbol will allow implementations to support various versions of POSIX.1 simultaneously. For instance, when either _POSIX_SOURCE is defined or _POSIX_C_SOURCE is defined as 1, the system should make visible the same namespace as permitted and required by IEEE Std 1003.1-1990. When _POSIX_C_SOURCE is defined, the state of _POSIX_SOURCE is completely irrelevant.

It is expected that C bindings to future POSIX standards and amendments will define new values for _POSIX_C_SOURCE, with each new value reserving the namespace for that new standard or amendment, plus all earlier POSIX standards. Using a single feature text macro for all standards rather than a separate macro for each standard furthers the goal of eventually combining all of the C bindings into one standard.

It is further intended that these feature test macros apply only to the headers specified by POSIX.1. Implementations are expressly permitted to make visible symbols not specified by this part of ISO/IEC 9945, within both POSIX.1 and other headers, under the control of feature test macros that are not defined by this part of ISO/IEC 9945.

The area of namespace pollution versus additions to structures is difficult because of the macro structure of C. The following discussion summarizes all the various problems with and objections to the issue.

Note the phrase “user defined macro.” Users are not permitted to define macro names (or any other name) beginning with _[A-Z_]. Thus, the conflict cannot occur for symbols reserved to the vendor's namespace, and the permission to add fields automatically applies, without qualification, to those symbols.

1) Data structures (and unions) need to be defined in headers by implementations to meet certain requirements of POSIX.1 and the C Standard {2}.
2) The structures defined by POSIX.1 are typically minimal, and any practical implementation would wish to add fields to these structures either to hold additional related information or for backwards compatibility (or both). Future standards (and de facto standards) would also wish to add to these structures. Issues of field alignment make it impractical (at least in the general case) to simply omit fields when they are not defined by the particular standard involved.
   Struct *dirent* is an example of such a minimal structure (although one could argue about whether the other fields need visible names). The *st_rdev* field of most implementations' *stat* structure is a common example where extension is needed and where a conflict could occur.
3) Fields in structures are in an independent namespace, so the addition of such fields presents no problem to the C language itself in that such names cannot interact with identically named user symbols because access is qualified by the specific structure name.
4) There is an exception to this: macro processing is done at a lexical level. Thus, symbols added to a structure might be recognized as user-provided macro names at the location where the structure is declared. This only can occur if the user-provided name is declared as a macro before the header declaring the structure is included. The user's use of the name after the declaration cannot interfere with the structure because the symbol is hidden and only accessible through access to the structure. Presumably, the user would not declare such a macro if there was an intention to use that field name.
5) Macros from the same or a related header might use the additional fields in the structure, and those field names might also collide with user macros. Although this is a less frequent occurrence, since macros are expanded at the point of use, no constraint on the order of use of names can apply.
6) An “obvious” solution of using names in the reserved namespace and then redefining them as macros when they should be visible does not work because this has the effect of exporting the symbol into the general namespace. For example, given a (hypothetical) system-provided header `<h.h>`, and two parts of a C program in `a.c` and `b.c`:
   In header `<h.h>`:
   ```
   struct foo {
           int __i;
   }
   #ifdef _FEATURE_TEST
   #define i __i;
   #endif
   ```
   In file `a.c`:
   ```
   #include h.h
   extern int i;
   ...
   ```
   In file `b.c`:
   ```
   extern int i;
   ...
   ```
   The symbol that the user thinks of as `i` in both files has an external name of `"__i"` in `a.c`; the same symbol i in `b.c` has an external name `"i"` (ignoring any hidden manipulations the compiler might perform on the names). This would cause a mysterious name resolution problem when `a.o` and `b.o` are linked.
   Simply avoiding definition then causes alignment problems in the structure.
   A structure of the form
   ```
   struct foo {
           union {
              int __i;
   #ifdef _FEATURE_TEST
              int i;
   ```

```
#endif
        } __ii;
}
```

does not work because the name of the logical field `i` is `"__ii.i"`, and introduction of a macro to restore the logical name immediately reintroduces the problem discussed previously (although its manifestation might be more immediate because a syntax error would result if a recursive macro did not cause it to fail first).

7) A more workable solution would be to declare the structure:

```
struct foo {
#ifdef _FEATURE_TEST
   int i;
#else
   int __i;
#endif
}
```

However, if a macro (particularly one required by a standard) is to be defined that uses this field, two must be defined: one that uses `i`, the other that uses `__i`. If more than one additional field is used in a macro and they are conditional on distinct combinations of features, the complexity goes up as 2^n.

All this leaves a difficult situation: vendors must provide very complex headers to deal with what is conceptually simple and safe: adding a field to a structure. It is the possibility of user-provided macros with the same name that makes this difficult.

Several alternatives were proposed that involved constraining the user's access to part of the namespace available to the user (as specified by the C Standard {2}). In some cases, this was only until all the headers had been included. There were two proposals discussed that failed to achieve consensus:

— Limiting it for the whole program.
— Restricting the use of identifiers containing only uppercase letters until after all system headers had been included. It was also pointed out that because macros might wish to access fields of a structure (and macro expansion occurs totally at point of use) restricting names in this way would not protect the macro expansion, and thus the solution was inadequate.

It was finally decided that reservation of symbols would occur, but as constrained.

The current wording also allows the addition of fields to a structure, but requires that user macros of the same name not interfere. This allows vendors to either:

— Not create the situation [do not extend the structures with user-accessible names or use the solution in (7) above] or
— Extend their compilers to allow some way of adding names to structures and macros safely.

There are at least two ways that the compiler might be extended: add new preprocessor directives that turn off and on macro expansion for certain symbols (without changing the value of the macro) and a function or lexical operation that suppresses expansion of a word. The latter seems more flexible, particularly because it addresses the problem in macros as well as in declarations.

The following seems to be a possible implementation extension to the C language that will do this: any token that during macro expansion is found to be preceded by three # symbols shall not be further expanded in exactly the same way as described for macros that expand to their own name as in section 3.8.3.4 of the C Standard {2}. A vendor may also wish to implement this as an operation that is lexically a function, which might be implemented as

```
#define __safe_name(x) ###x
```

Using a function notation would insulate vendors from changes in standards until such a functionality is standardized (if ever). Standardization of such a function would be valuable because it would then permit third parties to take advantage of it portably in software they may supply.

The symbols that are "explicitly permitted, but not required by this part of ISO/IEC 9945" include those classified below. (That is, the symbols classified below might, but are not required to, be present when _POSIX_C_SOURCE is defined to have the value 199309L.)

- Symbols in 2.8 and 2.9 that are defined to indicate support for options or limits that are constant at compile-time.
- Symbols in the namespace reserved for the implementation by the C Standard {2}.
- Symbols in a namespace reserved for a particular type of extension (e.g., type names ending with `_t` in `<sys/types.h>`).
- Additional members of structures or unions whose names do not reduce the namespace reserved for applications (see B.2.7.2).

The phrase "when that header is included" was chosen to allow any fine structure of auxiliary headers the implementor may choose to use, as long as the net result is as required.

There are several common environments available today where a feature test macro would be useful to applications programmers during the transition to standard-conforming environments from certain common historical environments. The symbols in Table B-1, derived from common porting bases and industry specifications are suggested.

Table B-1 —Suggested Feature Test Macros

Symbol	Description
_V7	Version 7
_BSD	General BSD systems
_BSD4_2	4.2BSD
_BSD4_3	4.3BSD
_SYSIII	System III
_SYSV	System V. 1, V.2
_SYSV3	System V.3
_XPG*n*	X/Open Portability Guide, Issue *n*
_USR GROUP	The 1984 /usr/group standard

Only symbols that are actually in the porting base or industry specification should be enabled by these symbols.

Feature test macros for implementation extensions will also probably be required. Quite a few of these are traditionally available, but are in violation of the intent of namespace pollution control. These can be made conforming simply by prefixing them with an underscore. Symbols beginning with "_POSIX" are strongly discouraged, as they will probably be used by later revisions of POSIX.1.

The environment for compilation has traditionally been fairly portable in historical systems, but during the transition to the C Standard {2} there will be confusion about how to specify that a C Standard {2} compiler is expected, as considerations of backwards compatibility will constrain many implementors from providing a conformant environment replacing the traditional one. This concern has more to do with the issues of namespace than with the syntax of the language accepted, which is highly compatible.

For systems that are sufficiently similar to traditional UNIX systems for this to make sense, it is suggested that if a compilation line of the form

```
        CC -D__STDC__ ...;
```

is provided, that the system provide an environment that is conformant with the C Standard {2}, at least with respect to namespace.

It was decided to use feature test macros, rather than the inclusion of a header, both because `<unistd.h>` was already in use and would itself have this problem, and because the underlying mechanism would probably have been this anyway, but in a less flexible fashion.

POSIX.1 requires that headers be included in all cases, although it is not directly clear from the text at this point in the standard. If a function does not need any special types, then it must be declared in `<unistd.h>`, as stated here. If it does require something special, then it has an associated header, and the program will not compile without that header.

B.2.7.3 Headers and Function Prototypes

The statement that names need not be carried forward literally exists for several reasons. These include the fact that some vendors may historically use other names and that the names are irrelevant to application portability. More importantly, because of the pervasive nature of C macros, a declaration of the form:

```
        kill (pid_t pid, int sig);
```

could be seriously undermined by a (perfectly valid) user declaration of the form:

```
        #define pid statusstruct.pidinfo
```

B.2.8 Numerical Limits

This subclause clarifies the scope and mutability of several classes of limits.

B.2.8.1 C Language Limits

See also 2.7 and B.1.1.1.

{CHAR_MIN}	It is possible to tell if the implementation supports native character comparison as signed or unsigned by comparing this limit to zero.
{WORD_BIT}	This limit has been omitted, as it is not referenced elsewhere in POSIX.1.

No limits are given in `<limits.h>` for floating point values because none of the functions in POSIX.1 use floating point values, and all the functions that do that are imported from the C Standard {2} by 8.1, as are the limits that apply to the floating point values associated with them.

Though limits to the addresses to system calls were proposed, they were not included in POSIX.1 because it is not clear how to implement them for the range of systems being considered, and no complete proposal was ever received. Limits regarding hardware register characteristics were similarly proposed and not attempted.

B.2.8.2 Minimum Values

There has been confusion about the minimum maxima, and when that is understood there is still a concern about providing ways to allocate storage based on the symbols. This is particularly true for those in 2.8.4 where an indeterminate value will leave the programmer with no symbol upon which to fall back.

Providing explicit symbols for the minima (from the implementor's point of view, or maxima from the the application's point of view) helps to resolve possible confusion. Symbols are still provided for the actual value, and it is expected that many applications will take advantage of these larger values, but they need not do so unless it is to their advantage. Where the values in this subclause are adequate for the application, it should use them. These are given symbolically both because it is easier to understand and because the values of these symbols could change between revisions of POSIX.1. Arguments to "good programming practice" also apply.

B.2.8.3 Run-Time Increasable Values

The heading of the far-right column of the table is given as "Minimum Value" rather than "Value" in order to emphasize that the numbers given in that column are minimal for the actual values a specific implementation is permitted to define in its `<limits.h>`. The values in the actual `<limits.h>` define, in turn, the maximum amount of a given resource that a Conforming POSIX.1 Application can depend on finding when translated to execute on that implementation. A Conforming POSIX.1 Application Using Extensions must function correctly even if the value given in `<limits.h>` is the minimum that is specified in POSIX.1. (The application may still be written so that it performs more efficiently when a larger value is found in `<limits.h>`.) A conforming implementation must provide at least as much of a particular resource as that given by the value in POSIX.1. An implementation that cannot meet this requirement (a "toy implementation") cannot be a conforming implementation.

B.2.8.4 Run-Time Invariant Values (Possibly Indeterminate)

{CHILD_MAX}

This name can be misleading. This limit applies to all processes in the system with the same user ID, regardless of ancestry.

{PTHREAD_KEYS_MAX}

The definition of this symbol is inconsistent with the motivations for defining the *pthread_key_delete*() function. It is expected that a future revision of this standard will redefine this symbol to be the maximum number of data keys that can exist in a process at one time.

B.2.8.5 Pathname Variable Values

{MAX_INPUT}

Since the only use of this limit is in relation to terminal input queues, it mentions them specifically. This limit was originally named {MAX_CHAR}. Application writers should use {MAX_INPUT} primarily as an indication of the number of bytes that can be written as a single unit by one Conforming POSIX.1 Application Using Extensions communicating with another via a terminal device. It is not implied that input lines received from terminal devices always contain {MAX_INPUT} bytes or fewer: an application that attempts to read more than {MAX_INPUT} bytes from a terminal may receive more than {MAX_INPUT} bytes.

It is not obvious that {MAX_INPUT} is of direct value to the application writer. The existence of such a value (whatever it may be) is directly of use in understanding how the tty driver works (particularly with respect to flow control and dropped characters). The value can be determined by finding out when flow control takes effect (see the description of IXOFF in 7.1.2.2).

Understanding that the limit exists and knowing its magnitude is important to making certain classes of applications work correctly. It is unlikely to be used in an application, but its presence makes POSIX.1 clearer.

{PATH_ MAX}

A Conforming POSIX.1 Application or Conforming POSIX.1 Application Using Extensions that, for example, compiles to use different, algorithms depending on the value of {PATH_MAX} should use code such as:

```
#if defined(PATH_MAX) && PATH_MAX < 512
```

```
       ...
#else
#if defined(PATH_MAX) /* PATH_MAX >= 512 */
       ...
#else /* PATH_MAX indeterminate */
        ...
#endif
#endif
```

This is because the value tends to be very large or indeterminate on most historical implementations (it is arbitrarily large on System V). On such systems there is no way to quantify the limit, and it seems counterproductive to include an artificially small fixed value in `<limits.h>` in such cases.

B.2.9 Symbolic Constants

B.2.9.1 Symbolic Constants for the *access*() Function

There is no additional rationale provided for this subclause.

B.2.9.2 Symbolic Constants for the *lseek*() Function

There is no additional rationale provided for this subclause.

B.2.9.3 Compile-Time Symbolic Constants for Portability Specifications

The purpose of this material is to allow an application developer to have a chance to determine whether a given application would run (or run well) on a given implementation. To this purpose has been added that of simplifying development of verification suites for POSIX.1. The constants given here were originally proposed for a separate file, `<posix.h>`, but it was decided that they should appear in `<unistd.h>` along with other symbolic constants.

As mentioned elsewhere in this rationale, thread-safe functions are useful beyond a multithreaded environment, thus the creation of the {_POSIX_THREAD_SAFE_FUNCTIONS} option.

If multithreaded applications are supported, thread-safe functions are necessary for correct operation of current application code.

B.2.9.4 Execution-Time Symbolic Constants for Portability Specifications

Without the addition of {_POSIX_NO_TRUNC} and {_PC_NO_TRUNC} to this list, POSIX.1 says nothing about the effect of a pathname component longer than {NAME_MAX}. There are only two effects in common use in implementations: truncation or an error. It is desirable to limit allowable behavior to these two cases. It is also desirable to permit applications to determine what an implementation's behavior is because services that are available with one behavior may be impractical to provide with the other. However, since the behavior may vary from one file system to another, it may be necessary to use *pathconf*() to resolve it.

B.3 Process Primitives

Consideration was given to enumerating all characteristics of a process defined by POSIX.1 and describing each function in terms of its effects on those characteristics, rather than English text. This is quite different from any known descriptions of historical implementations, and it was not certain that this could be done adequately and completely enough to produce a usable standard. Providing such descriptions in addition to the text was also considered. This was not done because it would provide at best two redundant descriptions, and more likely two descriptions with subtle inconsistencies.

B.3.1 Process Creation and Execution

Running a new program takes two steps. First the existing process (the parent) calls the *fork*() function, producing a new process (the child), which is a copy of itself. One of these processes (normally, but not necessarily, the child) then calls one of the *exec* functions to overlay itself with a copy of the new process image.

If the new program is to be run synchronously (the parent suspends execution until the child completes), the parent process then uses either the *wait*() or *waitpid*() function. If the new program is to be run asynchronously, it does not suffice to simply omit the *wait*() or *waitpid*() call, because after the child terminates it continues to hold some resources until it is waited for. A common way to produce ("spawn") a descendant process that does not need to be waited on is to *fork*() to produce a child and *wait*() on the child. The child *fork*()s again to produce a grandchild. The child then exits and the parent's *wait*() returns. The grandchild is thus disinherited by its grandparent.

A simpler method (from the programmer's point of view) of spawning is to do

```
system("something &");
```

However, this depends on features of a process (the shell) that are outside the scope of POSIX.1, although they are currently being addressed by the working group preparing ISO/IEC 9945-2 {B36} .

B.3.1.1 Process Creation

Many historical implementations have timing windows where a signal sent to a process group (e.g., an interactive SIGINT) just prior to or during execution of *fork*() is delivered to the parent following the *fork*() but not to the child because the *fork*() code clears the child's set of pending signals. POSIX.1 does not require, or even permit, this behavior. However, it is pragmatic to expect that problems of this nature may continue to exist in implementations that appear to conform to POSIX.1 and pass available verification suites. This behavior is only a consequence of the implementation failing to make the interval between signal generation and delivery totally invisible. From the application's perspective, a *fork*() call should appear atomic. A signal that is generated prior to the *fork*() should be delivered prior to the *fork*(). A signal sent to the process group after the *fork*() should be delivered to both parent and child. The implementation might actually initialize internal data structures corresponding to the child's set of pending signals to include signals sent to the process group during the *fork*(). Since the *fork*() call can be considered as atomic from the application's perspective, the set would be initialized as empty and such signals would have arrived after the *fork*(). See also B.3.3.1.2.

One approach that has been suggested to address the problem of signal inheritance across *fork*() is to add an [EINTR] error, which would be returned when a signal is detected during the call. While this is preferable to losing signals, it was not considered an optimal solution. Although it is not recommended for this purpose, such an error would be an allowable extension for an implementation.

The [ENOMEM] error value is reserved for those implementations that detect and distinguish such a condition. This condition occurs when an implementation detects that there is not enough memory to create the process. This is intended to be returned when [EAGAIN] is inappropriate because there can never be enough memory (either primary or secondary storage) to perform the operation. Because *fork*() duplicates an existing process, this must be a condition where there is sufficient memory for one such process, but not for two. Many historical implementations actually return [ENOMEM] due to temporary lack of memory, a case that is not generally distinct from [EAGAIN] from the perspective of a portable application.

Part of the reason for including the optional error [ENOMEM] is because the *SVID* {B41} specifies it and it should be reserved for the error condition specified there. The condition is not applicable on many implementations.

IEEE Std 1003.1-1988 neglected to require concurrent execution of the parent and child of *fork*(). A system that single-threads processes was clearly not intended and is considered an unacceptable, "toy implementation" of POSIX.1. The only objection anticipated to the phrase "executing independently" is testability, but this assertion should be testable.

Such tests require that both the parent and child can block on a detectable action of the other, such as a write to a pipe or a signal. An interactive exchange of such actions should be possible for the system to conform to the intent of POSIX.1.

The [EAGAIN] error exists to warn applications that such a condition might occur. Whether it will occur or not is not in any practical sense under the control of the application because the condition is usually a consequence of the user's use of the system, not of the application's code. Thus, no application can or should rely upon its occurrence under any circumstances, nor should the exact semantics of what concept of “user” is used be of concern to the application writer. Validation writers should be cognizant of this limitation.

There are two reasons why POSIX programmers call *fork*(). One reason is to create a new thread of control within the same program (which was originally only possible in POSIX by creating a new process); the other is to create a new process running a different program. In the latter case, the call to *fork*() is soon followed by a call to one of the *exec* functions.

The general problem with making *fork*() work in a multithreaded 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 will never be 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 multithreaded 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.

A cleaner and easier alternative is to define a single new operation that combines the *fork*() and *exec* functions and the miscellaneous code between them that sets up the new process state. This was considered to be too large a departure from standard practice.

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 (e.g., 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) will be reading the file descriptor after the *forkall*(). If this is not desired behavior, the parent process has to synchronize with such threads before calling *forkall*().

When *forkall*() is called, threads, other than the calling thread, that are in POSIX.1 functions that can return with an [EINTR] error may have those functions return [EINTR] if the implementation cannot ensure that the function will behave 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].

B.3.1.2 Execute a File

Early drafts of POSIX.1 required that the value of *argc* passed to *main*() be “one or greater.” This was driven by the same requirement in drafts of the C Standard {2}. 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 C Standard {2} and subsequently removed from POSIX.1 as well. The POSIX.1 wording, in particular the use of the word “should,” requires a Strictly Conforming POSIX.1 Application (see 1.3.3) 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.1 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."

Some systems can *exec* shell scripts. This functionality is outside the scope of POSIX.1, since it requires standardization of the command interpreter language of the script and/or where to find a command interpreter. These fall in the domain of the shell and utilities standard, currently under development as ISO/IEC 9945-2 {B36} . However, it is important that POSIX.1 neither require nor preclude any reasonable implementation of this behavior. In particular, the description of the [ENOEXEC] error is intended to permit discretion to implementations on whether to give this error for 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. 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.

Some implementations provide a third argument to *main*() called *envp*. This is defined as a pointer to the environment. The C Standard {2} specifies invoking *main*() with two arguments, so implementations must support applications written this way. Since POSIX.1 defines the global variable *environ*, which is also provided by historical implementations and can be used anywhere *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.

POSIX.1 specifies that signals set to SIG_IGN remain set to SIG_IGN and that the process signal mask be unchanged across an *exec*. This is consistent with historical implementations, and it permits some useful functionality, such as the nohup command. However, it should be noted that many existing applications wrongly assume that they start with certain signals set to the default action and/or unblocked. In particular, applications written with a simpler signal model that does not include blocking of signals, such as the one in the C Standard {2}, may not behave properly if invoked with some signals blocked. Therefore, it is best not to block or ignore signals across *execs* without explicit reason to do so, and especially not to block signals across *execs* of arbitrary (not closely co-operating) programs.

If {_POSIX_SAVED_IDS} is defined, 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 C Standard {2}, 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 C Standard {2}, summarizes the compatibility:

	dst:			
src:	**char *[]**	const **char*[]**	**char** *const[]	const **char***const[]
`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 nonexpert would naturally use would be that in the second row.

The C Standard {2} and POSIX.1 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 [e.g., *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 POSIX.1 definition 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 {ARC_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 nonconforming.

[ENAMETOOLONG]

Since the file pathname may be constructed by taking elements in the **PATH** variable and putting them together with the filename, the [ENAMETOOLONG] condition could also be reached this way.

[ETXTBSY] The error [ETXTBSY] was considered too implementation dependent to include. System V returns this error when the executable file is currently open for writing by some process. POSIX.1 neither requires nor prohibits this behavior.

Other systems (such as System V) may return [EINTR] from *exec*. This is not addressed by POSIX.1, 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].

B.3.1.3 Register Fork Handlers

There are at least two serious problems with the semantics of *fork*() in a multithreaded program. One problem has to do with state (e.g., 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 will be 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*() will call a *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 multithreaded libraries. Application programs may not be aware that a multithreaded library is in use, and they will 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 multithreaded library needs a way to protect its internal state during *fork*() in case it is reentered later in the child process. The problem arises especially in multithreaded 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 multithreaded libraries with a means to protect themselves from innocent application programs that call *fork*(), and it provides multithreaded 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 (e.g., 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.

B.3.2 Process Termination

Early drains drew a different distinction between normal and abnormal process termination. Abnormal termination was caused only by certain signals and resulted in implementation-defined “actions,” as discussed below. Subsequent drafts of POSIX.1 distinguished three types of termination: normal termination (as in the current POSIX.1), “simple abnormal termination,” and “abnormal termination with actions.” Again the distinction between the two types of abnormal termination was that they were caused by different signals and that implementation-defined actions would result in the latter case. Given that these actions were completely implementation defined, the early drafts were only saying when the actions could occur and how their occurrence could be detected, but not what they were. This was of little or no use to portable applications, and thus the distinction was dropped from POSIX.1.

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

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

Despite the silence of POSIX.1 on this feature, applications are advised not to create files named `core` because of potential conflicts in many implementations. Some historical implementations use a different name than `core` for the file, such as by appending the process ID to the filename.

B.3.2.1 Wait for Process Termination

A call to the *wait*() or *waitpid*() function only returns status on an immediate child process of the calling process; i.e., 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 will affect the behavior of a *wait*() from the original parent process. Nothing in POSIX.1 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:

- To support job control (see B.3.3).
- To permit a nonblocking version of the *wait*() function.
- 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 4.3BSD. The interface uses the *options* argument, which is identical to an argument to *wait3*(). The WUNTRACED flag is used only in conjunction with job control on systems supporting the Job Control option. Its name comes from 4.3BSD 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 POSIX.1, 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*() will fail 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. POSIX.1 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 4.3BSD.

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 earlier draft 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 POSIX.1 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 *ptraced* 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.

B.3.2.2 Terminate a Process

Most C language programs should use the *exit*() function rather than *_exit*(). The *_exit*() function is defined here instead of *exit*() because the C Standard {2} defines the latter to have certain characteristics that are beyond the scope of POSIX.1, specifically the flushing of buffers on open files and the use of *atexit*(). See “The C Language” in the Introduction. There are several public-domain implementations of *atexit*() that may be of use to interface implementors who wish to incorporate it.

It is important that the consequences of process termination as described in this subclause occur regardless of whether the process called *_exit*() [perhaps indirectly through *exit*()] or instead was terminated due to a signal or for some other reason. Note that in the specific case of *exit*() this means that the *status* argument to *exit*() is treated the same as the *status* argument to *_exit*(). See also B.3.2.

A language other than C may have other termination primitives than the C language *exit*() function, and programs written in such a language should use its native termination primitives, but those should have as part of their function the behavior of *_exit*() as described in this subclause. Implementations in languages other than C are outside the scope of the present version of POSIX.1, however.

As required by the C Standard {2}, using `return` from *main*() is equivalent to calling *exit*() with the same argument value. Also, reaching the end of the *main*() function is equivalent to using *exit*() with an unspecified value.

A value of zero (or EXIT_SUCCESS, which is required by 8.1 to be zero) for the argument *status* conventionally indicates successful termination. This corresponds to the specification for *exit*() in the C Standard {2}. 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-dependent 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 4.2BSD, 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 4.2BSD, 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 4.2BSD, 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 will receive similar treatment at that time. In POSIX.1, the signals will be sent to the entire process group at the same time. Also, in POSIX.1, but not in 4.2BSD, 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. POSIX.1 does not require sending SIGHUP and SIGCONT in those cases, because, unlike process termination, those cases will not be 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 3.3.1.2.

B.3.3 Signals

Signals, as defined in Version 7, System III, the *1984/ usr/group Standard* {B75} , and System V (except very recent releases), have shortcomings that make them unreliable for many application uses. Several objections were raised against early drafts of POSIX.1 because of this. Therefore, a new signal mechanism, based very closely on the one of 4.2BSD and 4.3BSD, was added to POSIX.1. With the exception of one feature [see item (4) below and also *sigpending*()], it is possible to implement the POSIX.1 interface as a simple library veneer on top of 4.3BSD. There are also a few minor aspects of the underlying 4.3BSD implementation (as opposed to the interface) that would also need to change to conform to POSIX.1.

The major differences from the BSD mechanism are:

1) *Signal mask type*. BSD uses the type *int* to represent a signal mask, thus limiting the number of signals to the number of bits in an *int* (typically 32.) The new standard instead uses a defined type for signal masks. Because of this change, the interface is significantly different than it is in BSD implementations, although the functionality, and potentially the implementation, are very similar.
2) *Restarting system calls*. Unlike all previous historical implementations, 4.2BSD restarts some interrupted system calls rather than returning an error with *errno* set to [EINTR] after the signal-catching function returns. This change caused problems for some existing application code. 4.3BSD and other systems derived from 4.2BSD allow the application to choose whether system calls are to be restarted. POSIX.1 (in 3.3.4) does not require restart of functions because it was not clear that the semantics of system-call restart in any historical implementation were useful enough to be of value in a standard. Implementors are free to add such mechanisms as extensions.
3) *Signal stacks*. The 4.2BSD mechanism includes a function *sigstack*(). The 4.3BSD mechanism includes this and a function *sigreturn*(). No equivalent is included in POSIX.1 because these functions are not portable, and no sufficiently portable and useful equivalent has been identified. See also 8.3.1.
4) *Pending signals*. The *sigpending*() function is the sole new signal operation introduced in POSIX.1.

A proposal was considered for making reliable signals optional. However, the consensus was that this would hurt application portability, as a large percentage of applications using signals can be hurt by the unreliable aspects of historical implementations of the *signal*() mechanism defined by the C Standard {2}. This unreliability stems from the fact that the signal action is reset to SIG_DFL before the user's signal-catching routine is entered. The C Standard {2} does not require this behavior, but does explicitly permit it, and most historical implementations behave this way.

For example, an application that catches the SIGINT signal using *signal*() could be terminated with no chance to recover when two such signals arrive sufficiently close in time (e.g., when an impatient user types the INTR character twice in a row on a busy system). Although the C Standard {2} no longer requires this unreliable behavior, many historical implementations, including System V, will reset the signal action to SIG_DFL. For this reason, it is strongly recommended that the *signal*() function not be used by POSIX.1 conforming applications. Implementations should also consider blocking signals during the execution of the signal-catching function instead of resetting the action to SIG_DFL, but backward compatibility considerations will most likely prevent this from becoming universal.

Most historical implementations do not queue signals; i.e., a process's signal handler is invoked once, even if the signal has been generated multiple times before it is delivered. A notable exception to this is SIGCLD, which, in System V, is queued. The queueing of signals is neither required nor prohibited by POSIX.1. See 3.3.1.2. It is expected that a future realtime extension to POSIX.1 will address the issue of reliable queueing of event notification.

Realtime Signals Extension

This portion of the rationale presents models, requirements, and standardization issues relevant to the Realtime Signals Extension. This extension provides the capability required to support reliable, deterministic, asynchronous notification of events. While a new mechanism, unencumbered by the historical usage and semantics of POSIX.1 signals, might allow for a more efficient implementation, the application requirements for event notification can be met with a small number of extensions to signals. Therefore, a minimal set of extensions to signals to support the application requirements is specified.

The realtime signal extensions specified in this clause are used by other realtime functions requiring asynchronous notification.

Models

The model supported is one of multiple cooperating processes, each of which handles multiple asynchronous external events. Events represent occurrences that are generated as the result, of some activity in the system. Examples of occurrences that can constitute an event include

- Completion of an asynchronous I/O request
- Expiration of a POSIX.1b timer
- Arrival of an interprocess message
- Generation of a user-defined event

Processing of these events may occur synchronously via polling for event notifications or asynchronously via a software interrupt mechanism. Existing practice for this model is well established for traditional proprietary realtime operating systems, realtime executives, and realtime extended POSIX-like systems.

A contrasting model is that of “cooperating sequential processes” where each process handles a single priority of events via polling. Each process blocks while waiting for events, and each process depends on the preemptive, priority-based process scheduling mechanism to arbitrate between events of different priority that need to be processed concurrently. Existing practice for this model is also well established for small realtime executives that typically execute in an unprotected physical address space, but it is just emerging in the context of a fuller function operating system with multiple virtual address spaces.

It could be argued that the cooperating sequential process model, and the facilities supported by the POSIX Threads Extension obviate a software interrupt model. But, even with the cooperating sequential process model, the need has been recognized for a software interrupt model to handle exceptional conditions and process aborting, so the mechanism must be supported in any case. Furthermore, it is not the purview of this standard to attempt to convince realtime practitioners that their current application models based on software interrupts are “broken” and should be replaced by the cooperating sequential process model. Rather, it is the charter of this standard to provide standard extensions to mechanisms that support existing realtime practice.

Requirements

This subclause discusses the following realtime application requirements for asynchronous event notification:

— Reliable delivery of asynchronous event notification
The events notification mechanism shall guarantee delivery of an event notification. Asynchronous operations (such as asynchronous I/O and timers) that complete significantly after they are invoked have to guarantee that delivery of the event notification can occur at the time of completion.
— Prioritized handling of asynchronous event notifications
The events notification mechanism shall support the assigning of a user function as an event notification handler. Furthermore, the mechanism shall support the preemption of an event handler function by a higher priority event notification and shall support the selection of the highest priority pending event notification when multiple notifications (of different priority) are pending simultaneously.
The model here is based on hardware interrupts. Asynchronous event handling allows the application to ensure that time-critical events are immediately processed when delivered, without the indeterminism of being at a random location within a polling loop. Use of handler priority allows the specification of how handlers are interrupted by other higher priority handlers.
— Differentiation between multiple occurrences of event notifications of the same type
The events notification mechanism shall pass an application-defined value to the event handler function. This value can be used for a variety of purposes, such as enabling the application to identify which of several possible events of the same type (for example, timer expirations) has occurred.
— Polled reception of asynchronous event notifications
The events notification mechanism shall support blocking and nonblocking polls for asynchronous event notification.
The polled mode of operation is often preferred over the interrupt mode by those practitioners accustomed to this model. Providing support for this model facilitates the porting of applications based on this model to POSIX.1b conforming systems.
— Deterministic response to asynchronous event notifications
The events notification mechanism shall not preclude implementations that provide deterministic event dispatch latency and shall minimize the number of system calls needed to use the event facilities during realtime processing.

Rationale for Extension

POSIX.1 signals have many of the characteristics necessary to support the asynchronous handling of event notifications, and the Realtime Signals Extension addresses the following deficiencies in the POSIX.1 signal mechanism:

— Signals do not support reliable delivery of event notification. Subsequent occurrences of a pending signal are not guaranteed to be delivered.
— Signals do not support prioritized delivery of event notifications. The order of signal delivery when multiple unblocked signals are pending is undefined.
— Signals do not support the differentiation between multiple signals of the same type.

B.3.3.1 Signal Concepts

B.3.3.1.1 Signal Names

The restriction on the actual type used for *sigset_t* is intended to guarantee that these objects can always be assigned, have their address taken, and be passed as parameters by value. It is not intended that this type be a structure including pointers to other data structures, as that could impact the portability of applications performing such operations. A reasonable implementation could be a structure containing an array of some integer type.

The signals described in POSIX.1 must have unique values so that they may be named as parameters of case statements in the body of a C language switch clause. However, implementation-defined signals may have values that overlap with each other or with signals specified in this document. An example of this is SIGABRT, which traditionally overlaps some other signal, such as SIGIOT.

SIGKILL, SIGTERM, SIGUSR1, and SIGUSR2 are ordinarily generated only through the explicit use of the *kill*() function, although some implementations generate SIGKILL under extraordinary circumstances. SIGTERM is traditionally the default signal sent by the kill command.

The signals SIGBUS, SIGEMT, SIGIOT, SIGTRAP, and SIGSYS were omitted from POSIX.1 because their behavior is implementation dependent and could not be adequately categorized. Conforming implementations may deliver these signals, but must document the circumstances under which they are delivered and note any restrictions concerning their delivery. The signals SIGFPE, SIGILL, and SIGSEGV are similar in that they also generally result only from programming errors. They were included in POSIX.1 because they do indicate three relatively well-categorized conditions. They are all defined by the C Standard {2} and thus would have to be defined by any system with a C Standard {2} binding, even if not explicitly included in POSIX.1.

There is very little that a Conforming POSIX.1 Application can do by catching, ignoring, or masking any of the signals SIGILL, SIGTRAP, SIGIOT, SIGEMT, SIGBUS, SIGSEGV, SIGSYS, or SIGFPE. They will generally be generated by the system only in cases of programming errors. While it may be desirable for some robust code (e.g., a library routine) to be able to detect and recover from programming errors in other code, these signals are not nearly sufficient for that purpose. One portable use that does exist for these signals is that a command interpreter can recognize them as the cause of a process's termination [with *wait*()] and print an appropriate message. The mnemonic tags for these signals are derived from their PDP-11 origin.

The signals SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU, and SIGCONT are provided for job control and are unchanged from 4.2BSD. The signal SIGCHLD is also typically used by job control shells to detect children that have terminated or, as in 4.2BSD, stopped. See also B.3.3.4.

Some implementations, including System V, have a signal named SIGCLD, which is similar to SIGCHLD in 4.2BSD. POSIX.1 permits implementations to have a single signal with both names. POSIX.1 carefully specifies ways in which portable applications can avoid the semantic differences between the two different implementations. The name SIGCHLD was chosen for POSIX.1 because most current application usages of it can remain unchanged in conforming applications. SIGCLD in System V has more cases of semantics that POSIX.1 does not specify, and thus applications using it are more likely to require changes in addition to the name change.

Some implementations that do not support Job Control may nonetheless implement SIGCHLD. Similarly, such an implementation may choose to implement SIGSTOP. Since POSIX.1 requires that symbolic names always be defined (with the exception of certain names in `<limits.h>` and `<unistd.h>`), a portable method of determining, at run-time, whether an optional signal is supported is to call the *sigaction*() function with **NULL** *act* and *oact* arguments. A successful return indicates that the signal is supported. Note that if *sysconf*() shows that Job Control is present, then all of the optional signals shall also be supported.

The signals SIGUSR1 and SIGUSR2 are commonly used by applications for notification of exceptional behavior and are described as “reserved as application defined” so that such use is not prohibited. Implementations should not generate SIGUSR1 or SIGUSR2, except when explicitly requested by *kill*(). It is recommended that libraries not use these two signals, as such use in libraries could interfere with their use by applications calling the libraries. If such use is unavoidable, it should be documented. It is prudent for nonportable libraries to use nonstandard signals to avoid conflicts with use of standard signals by portable libraries.

There is no portable way for an application to catch or ignore nonstandard signals. Some implementations define the range of signal numbers, so applications can install signal-catching functions for all of them. Unfortunately, implementation-defined signals often cause problems when caught or ignored by applications that do not understand the reason for the signal. While the desire exists for an application to be more robust by handling all possible signals [even those only generated by *kill*()], no existing mechanism was found to be sufficiently portable to include in POSIX.1. The value of such a mechanism, if included, would be diminished given that SIGKILL would still not be catchable.

A number of new signal numbers are reserved for applications because the two user signals defined by POSIX.1 are insufficient for many realtime applications. A range of signal numbers is specified, rather than an enumeration of additional reserved signal names, because different applications and application profiles will require a different number of application signals. It is not desirable to burden all application domains and therefore all implementations with the maximum number of signals required by all possible applications. Note that in this context, signal numbers are essentially different signal priorities.

The relatively small number of required additional signals, {_POSIX_RTSIG_MAX}, was chosen so as not to require an unreasonably large signal mask/set. While this number of signals defined in POSIX.1 will fit in a single 32 b word signal mask, it is recognized that most existing implementations define many more signals than are specified in POSIX.1 and, in fact, many implementations have already exceeded 32 signals (including the “null signal”). Support of {_POSIX_RTSIG_MAX} additional signals may push some implementation over the single 32 b word line, but is unlikely to push any implementations that are already over that line beyond the 64-signal line.

B.3.3.1.2 Signal Generation and Delivery

The terms defined in this subclause are not used consistently in documentation of historical systems. Each signal can be considered to have a lifetime beginning with *generation* and ending with *delivery* or *acceptance*. The POSIX.1 definition of *delivery* does not exclude ignored signals; this is considered a more consistent definition. This revised text in several parts of this standard clarifies the distinct semantics of asynchronous signal *delivery* and synchronous signal *acceptance*. The previous wording attempted to categorize both under the term *delivery*, which led to conflicts over whether the effects of asynchronous signal delivery applied to synchronous signal acceptance.

Signals generated for a process are delivered to only one thread. Thus, if more than one thread is eligible to receive a signal, one has to be chosen. The choice of threads is left entirely up to the implementation both to allow the widest possible range of conforming implementations and to give implementations the freedom to deliver the signal to the “easiest possible” thread should there be differences in ease of delivery between different threads.

Note that should multiple delivery among cooperating threads be required by an application, this can be trivially constructed out of the provided single-delivery semantics. The construction of a *sigwait_multiple*() function that accomplishes this goal is presented with the *sigwait* rationale.

Implementations should deliver unblocked signals as soon after they are generated as possible. However, it is difficult for POSIX.1 to make specific requirements about this, beyond those in *kill*() and *sigprocmask*(). Even on systems with prompt delivery, scheduling of higher priority processes is always likely to cause delays.

In general, the interval between the generation and delivery of unblocked signals cannot be detected by an application. Thus, references to pending signals generally apply to blocked, pending signals. An implementation registers a signal as pending on the process when no thread has the signal unblocked and there are no threads blocked in a *sigwait* function for that signal. Thereafter, the implementation delivers the signal to the first thread that unblocks the signal or calls a *sigwait* function on a signal set containing this signal rather than choosing the recipient thread at the time the signal is sent.

In the 4.3BSD system, signals that are blocked and set to SIG_IGN are discarded immediately upon generation. For a signal that is ignored as its default action, if the action is SIG_DFL and the signal is blocked, a generated signal remains pending. In the 4.1BSD system and in System V Release 3, two other implementations that support a somewhat similar signal mechanism, all ignored, blocked signals remain pending if generated. Because it is not normally useful for an application to simultaneously ignore and block the same signal, it was unnecessary for POSIX.1 to specify behavior that would invalidate any of the historical implementations.

There is one case in some historical implementations where an unblocked, pending signal does not remain pending until it is delivered. In the System V implementation of *signal*(), pending signals are discarded when the action is set to SIG_DFL or a signal-catching routine (as well as to SIG_IGN). Except in the case of setting SIGCHLD to SIG_DFL, implementations that do this do not conform completely to POSIX.1. Some earlier drafts of POSIX.1 explicitly stated this, but these statements were redundant due to the requirement that functions defined by POSIX.1 not change attributes of processes defined by POSIX.1 except as explicitly stated (see Section 3).

POSIX.1 specifically states that the order in which multiple, simultaneously pending signals are delivered is unspecified. This order has not been explicitly specified in historical implementations, but has remained quite consistent and been known to those familiar with the implementations. Thus, there have been cases where applications (usually system utilities) have been written with explicit or implicit dependencies on this order. Implementors and others porting existing applications may need to be aware of such dependencies.

When there are multiple pending signals that are not blocked, implementations should arrange for the delivery of all signals at once, if possible. Some implementations stack calls to all pending signal-catching routines, making it appear that each signal-catcher was interrupted by the next signal. In this case, the implementation should ensure that this stacking of signals does not violate the semantics of the signal masks established by *sigaction*(). Other implementations process at most one signal when the operating system is entered, with remaining signals saved for later delivery. Although this practice is widespread, this behavior is neither standardized nor endorsed. In either case, implementations should attempt to deliver signals associated with the current state of the process (e.g., SIGFPE) before other signals, if possible.

In 4.2BSD and 4.3BSD, it is not permissible to ignore or explicitly block SIGCONT because if blocking or ignoring this signal prevented it from continuing a stopped process, such a process could never be continued (only killed by SIGKILL). However, 4.2BSD and 4.3BSD do block SIGCONT during execution of its signal-catching function when it is caught, creating exactly this problem. A proposal was considered to disallow catching SIGCONT SIGCONT in addition to ignoring and blocking it, but this limitation led to objections. The consensus was to require that SIGCONT always continue a stopped process when generated. This removed the need to disallow ignoring or explicit blocking of the signal; note that SIG_IGN and SIG_DFL are equivalent for SIGCONT.

The Realtime Signals Extension to POSIX.1 signal generation and delivery behavior is required for the following reasons:

1) The *sigevent* structure is used by other POSIX.1 functions that result in asynchronous event notifications to specify the notification mechanism to use and other information needed by the notification mechanism. The standard defines only three symbolic values for the notification mechanism. SIGEV_NONE is used to indicate that no notification is required when the event occurs. This is useful for applications that use asynchronous I/O with polling for completion. SIGEV_SIGNAL indicates that a signal shall be generated when the event occurs. SIGEV_NOTIFY provides for "callback functions" for asynchronous notifications done by a function call within the context of a new thread. This provides a multithreaded process a more natural means of notification than signals. The primary difficulty with previous notification approaches has been to specify the environment of the notification routine.
    a) One approach is to limit the notification routine to call only functions permitted in a signal handler. While the list of permissible functions is clearly stated, this is overly restrictive.

b) A second approach is to define a new list of functions or classes of functions that are explicitly permitted or not permitted. This would give a programmer more lists to deal with, which would be awkward.

c) The third approach is to define completely the environment for execution of the notification function. A clear definition of an execution environment for notification is provided by executing the notification function in the environment of a newly created thread.

Implementations may support additional notification mechanisms by defining new values for *sigev_notify*.

For a notification type of SIGEV_SIGNAL, the other members of the *sigevent* structure defined by the standard specify the realtime signal—that is, the signal number and application-defined value that differentiates between occurrences of signals with the same number—that will be generated when the event occurs. The structure is defined in `<signal.h>`, even though the structure is not directly used by any of the signal functions, because it is part of the signals interface used by the POSIX.1b "client functions." When the client functions include `<signal.h>` to define the signal names, the *sigevent* structure will also be defined.

An application-defined value passed to the signal handler is used to differentiate between different "events" instead of requiring that the application use different signal numbers for several reasons:

a) Realtime applications potentially handle a very large number of different events. Requiring that implementations support a correspondingly large number of distinct signal numbers will adversely impact the performance of signal delivery because the signal masks to be manipulated on entry and exit to the handlers will become large.

b) Event notifications are prioritized by signal number (the rationale for this is explained in the following paragraphs) and the use of different signal numbers to differentiate between the different event notifications overloads the signal number more than has already been done. It also requires that the application writer make arbitrary assignments of priority to events that are logically of equal priority.

A union is defined for the application-defined value so that either an integer constant or a pointer can be portably passed to the signal-catching function. On some architectures a pointer cannot be cast to an *int* and vice versa.

Use of a structure here with an explicit notification type discriminant rather than explicit parameters to realtime functions, or embedded in other realtime structures, provides for future extensions to the standard. Additional, perhaps more efficient, notification mechanisms can be supported for existing realtime function interfaces, such as timers and asynchronous I/O, by extending the *sigevent* structure appropriately. The existing realtime function interfaces will not have to be modified to use any such new notification mechanism. The revised text concerning the SIGEV_SIGNAL value makes consistent the semantics of the members of the *sigevent* structure, particularly in the definitions of *lio_listio*() and *aio_fsync*(). For uniformity, other revisions cause this specification to be referred to rather than inaccurately duplicated in the descriptions of functions and structures using the *sigevent* structure. The revised wording does not relax the requirement in 6.7.1.1 that the signal number be in the range SIGRTMIN to SIGRTMAX to guarantee queueing and passing of the application value, since that requirement is still implied by 3.3.1.1.

2) The standard is intentionally vague on whether "nonrealtime" signal-generating mechanisms can result in a *siginfo_t* being supplied to the handler on delivery. In one existing implementation, a *siginfo_t* is posted on signal generation, even though the implementation does not support queuing of multiple occurrences of a signal. It is not the intent of the standard to preclude this, independent of the mandate to define signals that do support queuing. Any interpretation that appears to preclude this is a mistake in the reading or writing of the standard.

3) Signals handled by realtime signal handlers might be generated by functions or conditions that do not allow the specification of an application-defined value and do not queue. The standard specifies the *si_code* member of the *siginfo_t* structure used in existing practice and defines additional codes so that applications can detect whether an application-defined value is present or not. The code SI_USER for *kill*()-generated signals is adopted from existing practice.

4) The *sigaction*() *sa_flags* value SA_SIGINFO tells the implementation that the signal-catching function expects two additional arguments. When the flag is not set, a single argument, the signal number, is passed as specified by ISO/IEC 9945-1 : 1990. Although this part of ISO/IEC 9945 does not explicitly allow the *info* argument to the handler function to be **NULL,** this is existing practice. This provides for compatibility with programs whose signal-catching functions are not prepared to accept the additional arguments. The standard is explicitly unspecified as to whether signals actually queue when SA_SIGINFO is not set for a signal, as there appears to be no benefits to applications in specifying one behavior or another. One existing

implementation queues a *siginfo_t* on each signal generation, unless the signal is already pending, in which case the implementation discards the new *siginfo_t*; that is, the queue length is never greater than one. This implementation only examines SA_SIGINFO on signal delivery, discarding the queued *siginfo_t* if its delivery was not requested.

The standard specifies several new values for the *si_code* member of the *siginfo_t* structure. In existing practice, a *si_code* value of less than or equal to zero indicates that the signal was generated by a process via the *kill*() function. In existing practice, values of *si_code* that provide additional information for implementation-generated signals, such as SIGFPE or SIGSEGV, are all positive. Thus, if implementations define the new constants specified in this standard to be negative numbers, programs written to use existing practice will not break. This standard chose not to attempt to specify existing practice values of *si_code* other than SI_USER both because it was deemed beyond the scope of this standard and because many of the values in existing practice appear to be platform and implementation specific. But, the standard does specify that if an implementation—for example, one that does not have existing practice in this area—chooses to define additional values for *si_code*, these values have to be different from the values of the symbols specified by this standard. This will allow portable applications to differentiate between signals generated by one of the POSIX.1b asynchronous events and those generated by other implementation events in a manner compatible with existing practice.

The unique values of *si_code* for the POSIX.1b asynchronous events have implications for implementations of, for example, asynchronous I/O or message passing in user space library code. Such an implementation will be required to provide a hidden interface to the signal generation mechanism that allows the library to specify the standard values of *si_code*.

Existing practice also defines additional members of *siginfo_t*, such as the process ID and user ID of the sending process for *kill*()-generated signals. These members were deemed not necessary to meet the requirements of realtime applications and are not specified by the standard. Neither are they precluded.

The third argument to the signal-catching function, *context*, is left undefined by this standard, but is specified in the interface because it matches existing practice for the SA_SIGINFO flag. It was considered undesirable to require a separate implementation for SA_SIGINFO for POSIX conformance on implementations that already support the two additional parameters.

5) The requirement to deliver lower numbered signals in the range SIGRTMIN to SIGRTMAX first, when multiple unblocked signals are pending, results from several considerations:

   a) A method is required to prioritize event notifications. The signal number was chosen instead of, for instance, associating a separate priority with each request, because an implementation has to check pending signals at various points and select one for delivery when more than one is pending. Specifying a selection order is the minimal additional semantic that will achieve prioritized delivery. If a separate priority were to be associated with queued signals, it would be necessary for an implementation to search all nonempty, nonblocked signal queues and select from among them the pending signal with the highest priority. This would significantly increase the cost of and decrease the determinism of signal delivery.

   b) Given the specified selection of the lowest numeric unblocked pending signal, preemptive priority signal delivery can be achieved using signal numbers and signal masks by ensuring that the *sa_mask* for each signal number blocks all signals with a higher numeric value.

      For realtime applications that want to use only the newly defined realtime signal numbers without interference from the ISO/IEC 9945-1 : 1990 signals, this can be achieved by blocking all of the ISO/IEC 9945-1 : 1990 signals in the process signal mask and in the *sa_mask* installed by the signal action for the realtime signal handlers.

   The standard explicitly leaves unspecified the ordering of signals outside of the range of realtime signals and the ordering of signals within this range with respect to those outside the range. It was believed that this would unduly constrain implementations or standards in the future definition of new signals.

B.3.3.1.3 Signal Actions

Earlier drafts of POSIX.1 mentioned SIGCONT as a second exception to the rule that signals are not delivered to stopped processes until continued Because POSIX.1 now specifies that SIGCONT causes the stopped process to continue when it is generated, delivery of SIGCONT is not prevented because a process is stopped, even without an explicit exception to this rule.

Ignoring a signal by setting the action to SIG_IGN (or SIG_DFL for signals whose default action is to ignore)is not the same as installing a signal-catching function that simply returns. Invoking such a function will interrupt certain system functions that block processes [e.g., *wait*(), *sigsuspend*(), *pause*(), *read*(), *write*()] while ignoring a signal has no such effect on the process.

Historical implementations discard pending signals when the action is set to SIG_IGN. However, they do not always do the same when the action is set to SIG_DFL and the default action is to ignore the signal. POSIX.1 requires this for the sake of consistency and also for completeness, since the only signal this applies to is SIGCHLD, and POSIX.1 disallows setting its action to SIG_IGN.

The specification of the effects of SIG_IGN on SIGCHLD as implementation defined permits, but does not require, the System V effect of causing terminating children to be ignored by *wait*(). Yet it permits SIGCHLD to be effectively ignored in an implementation-independent manner by use of SIG_DFL.

Some implementations (System V, for example) assign different semantics for SIGCLD depending on whether the action is set to SIG_IGN or SIG_DFL. Since POSIX.1 requires that the default action for SIGCHLD be to ignore the signal, applications should always set; the action to SIG_DFL in order to avoid SIGCHLD.

Some implementations (System V, for example) will deliver a SIGCLD signal immediately when a process establishes a signal-catching function for SIGCLD when that process has a child that has already terminated. Other implementations, such as 4.3BSD, do not generate a new SIGCHLD signal in this way. In general, a process should not attempt to alter the signal action for the SIGCHLD signal while it has any outstanding children. However, it is not always possible for a process to avoid this; for example, shells sometimes start up processes in pipelines with other processes from the pipeline as children. Processes that cannot ensure that they have no children when altering the signal action for SIGCHLD thus need to be prepared for, but not depend on, generation of an immediate SIGCHLD signal.

The default action of the stop signals (SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU) is to stop a process that is executing. If a stop signal is delivered to a process that is already stopped, it has no effect. In fact, if a stop signal is generated for a stopped process whose signal mask blocks the signal, the signal will never be delivered to the process since the process must receive a SIGCONT, which discards all pending stop signals, in order to continue executing.

The SIGCONT signal shall continue a stopped process even if SIGCONT is blocked (or ignored). However, if a signal-catching routine has been established for SIGCONT, it will not be entered until SIGCONT is unblocked.

If a process in an orphaned process group stops, it is no longer under the control of a job-control shell and hence would not normally ever be continued. Because of this, orphaned processes that receive terminal-related stop signals (SIGTSTP, SIGTTIN, SIGTTOU, but not SIGSTOP) must not be allowed to stop. The goal is to prevent stopped processes from languishing forever. [As SIGSTOP is sent only via *kill*(), it is assumed that the process or user sending a SIGSTOP can send a SIGCONT when desired.] Instead, the system must discard the stop signal. As an extension, it may also deliver another signal in its place. 4.3BSD sends a SIGKILL, which is overly effective because SIGKILL is not catchable. Another possible choice is SIGHUP. 4.3BSD also does this for orphaned processes (processes whose parent has terminated) rather than for members of orphaned process groups; this is less desirable because job-control shells manage process groups. POSIX.1 also prevents SIGTTIN and SIGTTOU signals from being generated for processes in orphaned process groups as a direct result of activity on a terminal, preventing infinite loops when *read*() and *write*() calls generate signals that are discarded. (See B.7.1.1.4.) A similar restriction on the generation of SIGTSTP was considered, but that would be unnecessary and more difficult to implement due to its asynchronous nature.

Although POSIX.1 requires that signal-catching functions be called with only one argument, there is nothing to prevent conforming implementations from extending POSIX.1 to pass additional arguments, as long as Strictly

Conforming POSIX.1 Applications continue to compile and execute correctly. Most historical implementations do, in fact, pass additional, signal-specific arguments to certain signal-catching routines.

There was a proposal to change the declared type of the signal handler to:

```
void func (int sig, ...);
```

The usage of ellipses (", …") is C Standard {2} syntax to indicate a variable number of arguments. Its use was intended to allow the implementation to pass additional information to the signal handler in a standard manner.

Unfortunately, this construct would require all signal handlers to be defined with this syntax because the C Standard {2} allows implementations to use a different parameter passing mechanism for variable parameter lists than for nonvariable parameter lists. Thus, all existing signal handlers in all existing applications would have to be changed to use the variable syntax in order to be standard and portable. This is in conflict with the goal of Minimal Changes to Existing Application Code.

When terminating a process from a signal-catching function, processes should be aware of any interpretation that their parent may make of the status returned by *wait*() or *waitpid*(). In particular, a signal-catching function should not call *exit*(0) or *_exit*(0) unless it wants to indicate successful termination. A nonzero argument to *exit*() or *_exit*() can be used to indicate unsuccessful termination. Alternatively, the process can use *kill*() to send itself a fatal signal (first ensuring that the signal is set to the default action and not blocked). (See also B.3.2.2).

The behavior of *unsafe* functions, as defined by this subclause, is undefined when they are invoked from signal-catching functions in certain circumstances. The behavior of reentrant functions, as defined by this subclause, is as specified by POSIX.1, regardless of invocation from a signal-catching function. This is the only intended meaning of the statement that reentrant functions may be used in signal-catching functions without restriction. Applications must still consider all effects of such functions on such things as data structures, files, and process state. In particular, application writers need to consider the restrictions on interactions when interrupting *sleep*() [see *sleep*() and B.3.4.3] and interactions among multiple handles for a file description (see 8.2.3 and B.8.2.3). 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 nonreentrant function calls, applications should protect calls to these functions either by blocking the appropriate signals or through the use of some programmatic semaphore. POSIX.1 does not address the more general problem of synchronizing access to shared data structures. Note in particular that even the "safe" functions may modify the global variable *errno*; the signal-catching function may want to save and restore its value. The same principles apply to the reentrancy of application routines and asynchronous data access.

Note that *longjmp*() and *siglongjmp*() are not in the list of reentrant functions. This is because the code executing after *longjmp*() or *siglongjmp*() can call any unsafe functions with the same danger as calling those unsafe functions directly from the signal handler. Applications that use *longjmp*() or *siglongjmp*() out of signal handlers require rigorous protection in order to be portable. Many of the other functions that are excluded from the list are traditionally implemented using either the C language *malloc*() or *free*() functions or the C language standard I/O library, both of which traditionally use data structures in a nonreentrant manner. Because any combination of different functions using a common data structure can cause reentrancy problems, POSIX.1 does not define the behavior when any unsafe function is called in a signal handler that interrupts any unsafe function.

The only realtime extension to signal actions is the addition of the additional parameters to the signal-catching function. This extension has been explained and motivated in the previous subclause. In making this extension, though, developers of POSIX.1b ran into issues relating to C Standard {2} function prototypes. In response to input from the POSIX.1 standard developers, members were added to the *sigaction* structure to specify function prototypes for the newer signal-catching function specified by POSIX.1b. These members follow changes that are being made to POSIX.1. Note that the standard explicitly states that these fields may overlap so that a union can be defined. This will enable existing implementations of ISO/IEC 9945-1 : 1990 to maintain binary compatibility when these extensions are added.

The *siginfo_t* structure was adopted for passing the application-defined value to match existing practice, but the existing practice has no provision for an application-defined value, so this was added. Note that POSIX normally reserves the "_t" type designation for opaque types. The *siginfo_t* breaks with this convention to follow existing practice and thus promote portability. Standardization of the existing practice for the other members of this structure may be addressed in the future.

Although it is not explicitly visible to applications, there are additional semantics for signal actions implied by queued signals and their interaction with other POSIX.1b realtime functions. Specifically,

— It is not necessary to queue signals whose action is SIG_IGN.
— For implementations that support POSIX.1b timers, some interaction with the timer functions at signal delivery is implied to manage the timer over-run count (see 14.2.4).

B.3.3.1.4 Signal Effects on Other Functions

The most common behavior of an interrupted function after a signal-catching function returns is for the interrupted function to give an [EINTR] error. However, there are a number of specific exceptions, including *sleep*() and certain situations with *read*() and *write*().

The historical implementations of many functions defined by POSIX.1 are not interruptible, but delay delivery of signals generated during their execution until after they complete. This is never a problem for functions that are guaranteed to complete in a short (imperceptible to a human) period of time. It is normally those functions that can suspend a process indefinitely or for long periods of time [e.g., *wait*(), *pause*(), *sigsuspend*(), *sleep*(), or *read*()/*write*() on a slow device like a terminal] that are interruptible. This permits applications to respond to interactive signals or to set timeouts on calls to most such functions with *alarm*(). Therefore, implementations should generally make such functions (including ones defined as extensions) interruptible.

Functions not mentioned explicitly as interruptible may be so on some implementations, possibly as an extension where the function gives an [EINTR] error. There are several functions [e.g., *getpid*(), *getuid*()] that are specified as never returning an error, which can thus never be extended in this way.

B.3.3.2 Send a Signal to a Process

The semantics for permission checking for *kill*() differ between System V and most other implementations, such as Version 7 or 4.3BSD. The semantics chosen for POSIX.1 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 POSIX.1, although a conforming implementation could provide such an extension.

The implementation-defined processes to which a signal cannot be sent may include the scheduler or init.

Most historical implementations use `kill (-1, sig)` from a super-user process to send a signal to all processes (excluding system processes like `init`). This use of the *kill*() function is for administrative purposes only; portable applications should not send signals to processes about which they have no knowledge. In addition, there are semantic variations among different implementations that, because of the limited use of this feature, were not necessary to resolve by standardization. System V implementations also use `kill (-1, sig)` from a nonsuper-user process to send a signal to all processes with matching user IDs. This use was considered neither sufficiently widespread nor necessary for application portability to warrant inclusion in POSIX.1.

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*() interface make only the weaker guarantee in POSIX.1, because they only deliver one signal each time a process enters the kernel. Modifications to such implementations to support the *sigaction*() interface 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 POSIX.1 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 4.2BSD, the SIGCONT signal can be sent to any descendant process regardless of user-ID security checks. This allows a job-control shell to continue a job even if processes in the job have altered their user IDs (as in the su command). In keeping with the addition of the concept of sessions, similar functionality is provided by allowing the SIGCONT signal to be sent to any process in the same session, regardless of user-ID security checks. This is less restrictive than BSD in the sense that ancestor processes (in the same session) can now be the recipient. It is more restrictive than BSD in the sense that descendant processes that form new sessions are now subject to the user-ID checks. A similar relaxation of security is not necessary for the other job-control signals since those signals are typically sent by the terminal driver in recognition of special characters being typed; the terminal driver bypasses all security checks.

In secure implementations, a process may be restricted from sending a signal to a process having a different security label. In order to prevent the existence or nonexistence of a process from being used as a covert channel, such processes should appear nonexistent to the sender; i.e., [ESRCH] should be returned, rather than [EPERM], if *pid* refers only to such processes.

Existing implementations vary on the result of a *kill*() with *pid* indicating an inactive process (a terminated process that has not been waited for by its parent). Some indicate success on such a call (subject to permission checking), while others give an error of [ESRCH]. Since POSIX.1's definition of process lifetime 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.

B.3.3.3 Manipulate Signal Sets

The implementation of the *sigemptyset*() [or *sigfillset*()] functions could quite trivially clear (or set) all the bits in the signal set. Alternatively, it would be able 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 [e.g., 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 POSIX.1. Although it is outside the scope of POSIX.1 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 earlier drafts of POSIX.1 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.

B.3.3.4 Examine and Change Signal Action

Although POSIX.1 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*() will return 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 portable applications that catch SIGCHLD are expected to install signal-catching functions that repeatedly call the *waitpid*() function with the WNOHANG flag set, acting on each child for which status is returned, until *waitpid*() returns zero. If stopped children are not of interest, the use of the SA_NOCLDSTOP flag can prevent the overhead from invoking the signal-catching routine when they stop.

Some historical implementations also define other mechanisms for stopping processes, such as the *ptrace*() function. These implementations usually do not generate a SIGCHLD signal when processes stop due to this mechanism; however, that is beyond the scope of POSIX.1.

POSIX.1 requires that calls to *sigaction*() that supply a **NULL** *act* argument succeed, even in the case of signals that cannot be caught or ignored (i.e., 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*().

POSIX.1 requires that *sigaction*() properly save and restore a signal action set up by the C Standard {2} *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*().

POSIX.1b extends the *sigaction*() function as specified by ISO/IEC 9945-1 : 1990 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. This extension has been explained and motivated above.

The new members of the *sigaction* structure, explained and motivated above, are specified in this subclause.

B.3.3.5 Examine and Change Blocked Signals

When a process'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 B.3.3.2 for a discussion of the requirement on delivery of signals.

B.3.3.6 Examine Pending Signals

There is no additional rationale provided for this subclause.

B.3.3.7 Wait for a Signal

Normally, at the beginning of a critical code section, a specified set of signals is blocked using the *sigprocmask*() function. When the process 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.

B.3.3.8 Synchronously Accept a Signal

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 large scale transaction processing in which the overall throughput of an application and the ability to clearly track states are more important than the ability to minimize the response time of individual event handling.

It is possible to construct a signal-waiting macro function out of the realtime signal function mechanism defined in this supplement. 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 proposal therefore provides two signal-waiting function interfaces—one that waits indefinitely and one with a timeout—as part of the overall realtime signal interface specification.

The specification of an interface 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 interface has been included.

Note that the semantics of the *sigwaitinfo*() function are nearly identical to that of the *sigwait*() function defined by IEEE P1003.1c. 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 interfaces are being maintained because some implementations may choose to implement the threads extension interfaces 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.1c *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 interfaces 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:

- — Returning an error for multiple simultaneous calls to *sigwait* functions for the same signal.
- — One or more threads return.
- — All waiting threads return.

— 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 pthreads 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

1) A multi-way signal wait can be built from the single-way wait
2) 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
3) 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.

B.3.3.9 Queue a Signal to a Process

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*… namespace already carved out for signals was preferable to extending the interface to *kill*().

Such an interface became necessary when the put/get event function of the message queues was removed. It should be noted that the *sigqueue*() interface 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*()—to process groups, for example—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," the standard 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. The standard does not preclude this behavior, but an implementation that allocated queuing resources from a systemwide pool (with per-sender limits) and that leaves queued signals pending after the sender exits is also permitted.

B.3.3.10 Send a Signal to a Thread

The *pthread_kill*() function provides a mechanism for asynchronously directing a signal at a thread in the calling process. This could be used, for instance, by one thread to effect broadcast delivery of a signal to a set of threads.

Note that *pthread_kill*() only causes the signal to be handled in the context of the given thread; the signal action (termination or stopping) affects the process as a whole.

B.3.4 Timer Operations

B.3.4.1 Schedule Alarm

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 4.3BSD, very large values of the *seconds* argument are silently rounded down to an implementation-defined maximum value. This maximum is large enough (on the order of several months) that the effect is not noticeable.

Application writers should note that the type of the argument *seconds* and the return value of *alarm*() is *unsigned int*. That means that a Strictly Conforming POSIX.1 Application cannot pass a value greater than the minimum guaranteed value for {UINT_MAX}, which the C Standard {2} 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 int* or *int*.

There were two possible choices for alarm generation in multithreaded 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.

Application writers should be aware of possible interactions when the same process uses both the *alarm*() and *sleep*() functions [see *sleep*() and B.3.4.3].

B.3.4.2 Suspend Process Execution

Many common uses of *pause*() have timing windows. The scenario involves checking a condition related to a signal and, if the signal has not occurred, calling *pause*(). When the signal occurs between the check and the call to *pause*(), the process often blocks indefinitely. The *sigprocmask*() and *sigsuspend*() functions can be used to avoid this type of problem.

B.3.4.3 Delay Process Execution

There are two general approaches to the implementation of the *sleep*() function. One is to use the *alarm*() function to schedule a SIGALRM signal and then suspend the process waiting for that signal. The other is to implement an independent facility. POSIX.1 permits either approach.

In order to comply with the wording of the introduction to Section 3, that no primitive shall change a process attribute unless explicitly described by POSIX.1, an implementation using SIGALRM must carefully take into account any SIGALRM signal scheduled by previous *alarm*() calls, the action previously established for SIGALRM, and whether SIGALRM was blocked. If a SIGALRM has been scheduled before the *sleep*() would ordinarily complete, the *sleep*() must be shortened to that time and a SIGALRM generated (possibly simulated by direct invocation of the signal-catching function) before *sleep*() returns. If a SIGALRM has been scheduled after the *sleep*() would ordinarily complete, it must be rescheduled for the same time before *sleep*() returns. The action and blocking for SIGALRM must be saved and restored.

Historical implementations often implement the SIGALRM-based version using *alarm*() and *pause*(). One such implementation is prone to infinite hangups, as described in B.3.4.2. Another such implementation uses the C language *setjmp*() and *longjmp*() functions to avoid that window. That implementation introduces a different problem: when the SIGALRM signal interrupts a signal-catching function installed by the user to catch a different signal, the *longjmp*() aborts that signal-catching function. An implementation based on *sigprocmask*(), *alarm*(), and *sigsuspend*() can avoid these problems.

Despite all reasonable care, there are several very subtle, but detectable and unavoidable, differences between the two types of implementations. These are the cases mentioned in POSIX.1 where some other activity relating to SIGALRM takes place, and the results are stated to be unspecified. All of these cases are sufficiently unusual as not to be of concern to most applications.

(See also the discussion of the term “realtime” in B.3.4.1.)

Because *sleep*() can be implemented using *alarm*(), the discussion about alarms occurring early under B.3.4.1 applies to *sleep*() as well.

Application writers should note that the type of the argument *seconds* and the return value of *sleep*() is *unsigned int*. That means that a Strictly Conforming POSIX.1 Application cannot pass a value greater than the minimum guaranteed value for {UINT_MAX}, which the C Standard {2} 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 int* or *int*.

Scheduling delays may cause the process to return from the *sleep*() function significantly after the requested time. In such cases, the return value should be set to zero, since the formula (requested time minus the time actually spent) yields a negative number and *sleep*() returns an *unsigned int*.

B.4 Process Environment

B.4.1 Process Identification

B.4.1.1 Get Process and Parent Process IDs

There is no additional rationale provided for this subclause.

B.4.2 User Identification

B.4.2.1 Get Real User, Effective User, Real Group, and Effective Group IDs

There is no additional rationale provided for this subclause.

B.4.2.2 Set User and Group IDs

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 two capabilities are derived from System V. Without them, a program may have to run as super-user in order to perform the same functions, because super-user 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.

A process with appropriate privilege on a system with this saved ID capability establishes all relevant IDs to the new value, since this function is used to establish the identity of the user during login or su. Any change to this behavior would be dangerous since it involves programs that need to be trusted.

The behavior of 4.2BSD and 4.3BSD that allows setting the real ID to the effective ID is viewed as a value-dependent special case of appropriate privilege.

B.4.2.3 Get Supplementary Group IDs

The related function *setgroups*() is a privileged operation and therefore is not covered by POSIX.1.

As implied by the definition of supplementary groups, the effective group ID may appear in the array returned by *getgroups*() or it may be returned only by *getegid*(). Duplication may exist, but the application needs to call *getegid*() to be sure of getting all of the information. Various implementation variations and administrative sequences will cause the set of groups appearing in the result of *getgroups*() to vary in order and as to whether the effective group ID is included, even when the set of groups is the same (in the mathematical sense of “set”). (The history of a process and its parents could affect the details of result.)

Applications writers should note that {NGROUPS_MAX} is not necessarily a constant on all implementations.

B.4.2.4 Get User Name

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 ISO/IEC 9945-1 : 1990 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 nonzero 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.

B.4.3 Process Groups

B.4.3.1 Get Process Group ID

4.3BSD provides a *getpgrp*() function that returns the process group ID for a specified process. Although this function is used to support 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 4.3BSD *getpgrp*() has been omitted from POSIX.1.

B.4.3.2 Create Session and Set Process Group ID

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, POSIX.1 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.1 session containing a single POSIX.1 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.

B.4.3.3 Set Process Group ID for Job Control

The *setpgid*() function is used to group processes together for the purpose of signaling, placement in foreground or background, and other job-control actions. See B.2.2.2.

The *setpgid*() function is similar to the *setpgrp*() function of 4.2BSD, except that 4.2BSD allowed the specified new process group to assume any value. This presents certain security problems and is more flexible than necessary to support job control.

To provide tighter security, *setpgid*() only allows the calling process to join a process group already in use inside its session or create a new process group whose process group ID was equal to its process ID.

When a job-control shell spawns a new job, the processes in the job must be placed into a new process group via *setpgid*(). There are two timing constraints involved in this action:

1) The new process must be placed in the new process group before the appropriate program is launched via one of the *exec* functions.
2) The new process must be placed in the new process group before the shell can correctly send signals to the new process group.

To address these constraints, the following actions are performed: The new processes call *setpgid*() to alter their own process groups after *fork*() but before *exec*. This satisfies the first constraint. Under 4.3BSD, the second constraint is satisfied by the synchronization property of *vfork*(); that is, the shell is suspended until the child has completed the *exec*, thus ensuring that the child has completed the *setpgid*(). A new version of *fork*() with this same synchronization property was considered, but it was decided instead to merely allow the parent shell process to adjust the process group of its child processes via *setpgid*(). Both timing constraints are now satisfied by having both the parent shell and the child attempt to adjust the process group of the child process; it does not matter which succeeds first.

Because it would be confusing to an application to have its process group change after it began executing (i.e., after *exec*) and because the child process would already have adjusted its process group before this, the [EACCES] error was added to disallow this.

One nonobvious use of *setpgid*() is to allow a job-control shell to return itself to its original process group (the one in effect when the job-control shell was executed). A job-control shell does this before returning control back to its parent when it is terminating or suspending itself as a way of restoring its job control "state" back to what its parent would expect. (Note that the original process group of the job-control shell typically matches the process group of its parent, but this is not necessarily always the case.) See also B.7.2.4.

B.4.4 System Identification

B.4.4.1 System Name

The values of the structure members are not constrained to have any relation to the version of POSIX.1 implemented in the operating system. An application should instead depend on {_POSIX_VERSION} and related constants defined in 2.9.

POSIX.1 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 is specific to System III, System V, and related implementations, and it does not exist in Version 7 or 4.3BSD. The values it returns are set at system compile time in those historical implementations.

4.3BSD 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 length of the host name is limited to 31 characters in most implementations and the host ID is a 32-bit integer.

B.4.5 Time

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 4.3BSD, have mechanisms capable of returning more precise times [see the description of *gettimeofday*() in B.4.5.1]. A generalized timing scheme to unify these various timing mechanisms has been proposed but not adopted in POSIX.1.

B.4.5.1 Get System Time

Implementations in which *time_t* is a 32-bit signed integer (most historical implementations) will fail in the year 2038. This version of POSIX.1 does not address this problem. However, the use of the new *time_t* type is mandated in order to ease the eventual fix.

The use of the header `<time.h>`, instead of `<sys/types.h>`, allows compatibility with the C Standard {2}.

Many historical implementations (including Version 7) and the *1984 /usr/group Standard* {B75} use *long* instead of *time_t*. POSIX.1 uses the latter type in order to agree with the C Standard {2}.

4.3BSD includes *time*() only as an interface to the more flexible *gettimeofday*() function.

B.4.5.2 Get Process Times

The accuracy of the times reported is intentionally left unspecified to allow implementations flexibility in design, from uniprocessor to multiprocessor 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 will always be able to see the total times of all its descendants.

(See also the discussion of the term “real time” in B.3.4.1)

If the type *clock_t* is defined to be a signed 32-bit integer, it will overflow in somewhat more than a year if there 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. POSIX.1 permits an implementation to make the reference point for the returned value be the startup time of the process, rather than system startup 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.

B.4.6 Environment Variables

B.4.6.1 Environment Access

Additional functions *putenv*() and *clearenv*() were considered but rejected because they were considered to be more oriented towards system administration than ordinary application programs. This is being reconsidered for an amendment to POSIX.1 because uses from within an application have been identified since the decision was made.

It was proposed that this function is properly part of Section 8. It is an extension to a function in the C Standard {2}. Because this function should be available from any language, not just C, it appears here, to separate it from the material in Section 8, which is specific to the C binding. (The localization extensions to C are not, at this time, appropriate for other languages.)

B.4.7 Terminal Identification

The difference between *ctermid*() and *ttyname*() is that *ttyname*() must be passed a file descriptor and returns the pathname of the terminal associated with that file descriptor, while *ctermid*() returns a string (such as `/dev/tty`) that will refer to the controlling terminal if used as a pathname. Thus *ttyname*() is useful only if the process already has at least one file open to a terminal.

The historical value of *ctermid*() is `/dev/tty`; this is acceptable. The *ctermid*() function should not be used to determine if a process actually has a controlling terminal, but merely the name that would be used.

B.4.7.1 Generate Terminal Pathname

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.

Portable 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.

If the *ctermid*() function returns a pathname, access to the file is not guaranteed.

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 will be short, as some implementations have more than two pathname components before reaching a logical device name.

B.4.7.2 Determine Terminal Device Name

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 nonzero 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.

B.4.8 Configurable System Variables

This subclause 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 B.5.7 as well.

Although a portable application can run on all systems by never demanding more resources than the minimum values published in POSIX.1, 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 will make 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 POSIX.1.
2) International applications vendors occasionally require knowledge of the number of clock ticks per second. Without the facilities of this subclause, 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 lead 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 run-time by suitably testing the various interfaces 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 run-time facility is not meant to provide ever-changing values that applications will 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. POSIX.1 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. These latter are described in B.5.7.

B.4.8.1 Get Configurable System Variables

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 will return a negative value to show that this symbolic constant is not even defined in this case.

B.4.8.1.1 Special Symbol {CLK_TCK}

{CLK_TCK} appears in POSIX.1 for backwards compatibility with IEEE Std 1003.1-1988. Its use is obsolescent.

B.4.8.2 Get Password From User

The *getpass*() function was explicitly excluded from POSIX.1 because it was found that the name was misleading, and it provided no functionality that the user could not easily implement within POSIX.1. The implication of some form of security, which was not actually provided, exceeded the small gain in convenience.

B.5 Files and Directories

See *pathname resolution*.

The wording regarding the group of a newly created regular file, directory, or FIFO in *open*(), *mkdir*(), *mkfifo*(), respectively, defines the two acceptable behaviors in order to permit both the System V (and Version 7) behavior (in which the group of the new object is set to the effective group ID of the creating process) and the 4.3BSD behavior (in which the new object has the group of its parent directory). An application that needs a file to be created specifically in one or the other of the possible groups should use *chown*() to ensure the new group regardless of the style of groups the interface implements. Most applications will not and should not be concerned with the group ID of the file.

B.5.1 Directories

Historical implementations prior to 4.2BSD had no special functions, types, or headers for directory access. Instead, directories were read with *read*() and each program that did so had code to understand the internal format of directory files. Many such programs did not correctly handle the case of a maximum-length (historically fourteen character) filename and would neglect to add a null character string terminator when doing comparisons. The access methods in POSIX.1 eliminate that bug, as well as hiding differences in implementations of directories or file systems.

The directory access functions originally selected for POSIX.1 were derived from 4.2BSD, were adopted in System V Release 3, and are in *SVID* {B41} Volume 3, with the exception of a type difference for the *d_ino* field. That field represents implementation-dependent or even file system-dependent information (the i-node number in most implementations). Since the directory access mechanism is intended to be implementation-independent, and since only system programs, not ordinary applications, need to know about the i-node number (or file serial number) in this context, the *d_ino* field does not appear in POSIX.1. Also, programs that want this information can get it with *stat*().

B.5.1.1 Format of Directory Entries

Information similar to that in the header `<dirent.h>` is contained in a file `<sys/dir.h>` in 4.2BSD and 4.3BSD. The equivalent in these implementations of *struct dirent* from POSIX.1 is *struct direct*. The filename was changed because the name `<sys/dir.h>` was also used in earlier implementations to refer to definitions related to the older access method; this produced name conflicts. The name of the structure was changed because POSIX.1 does not completely define what is in the structure, so it could be different on some implementations from *struct direct.*

The name of an array of *char* of an unspecified size should not be used as an *lvalue*. Use of

```
        sizeof (d_name)
```

is incorrect; use

```
        strlen (d_name)
```

instead.

The array of *char d_name* is not a fixed size. Implementations may need to declare *struct dirent* with an array size for *d_name* of 1, but the actual number of characters provided matches (or only slightly exceeds) the length of the file name.

Currently, implementations are excluded if they have *d_name* with type *char* *. Lacking experience of such implementations, the developers of POSIX.1 declined to try to describe in standards language what to do if either type were permitted.

B.5.1.2 Directory Operations

Based on historical implementations, the rules about file descriptors apply to directory streams as well. However, POSIX.1 does not mandate that the directory stream be implemented using file descriptors. The description of *opendir*() 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 returned value of *readdir*() merely *represents* a directory entry. No equivalence should be inferred.

The directory entries for dot and dot-dot are optional. POSIX.1 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, POSIX.1 imposes no portability requirements for erroneous program constructs, erroneous data, or the use of indeterminate values such as the use or referencing of a *dirp* value or a *dirent* structure value after a directory stream has been closed or alter a *fork*() or one of the *exec* function calls.

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 militate against the historical performance-oriented implementations.

Since *readdir*() returns **NULL** both:

1) When it detects an error, and
2) 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. Because the function must not change *errno* in case (2) and must set it to a nonzero value in case (1), 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.

Historical implementations include two more functions:

```
long telldir (dirp)
DIR *dirp;
void seekdir (dirp, loc)
DIR *dirp;
long loc;
```

The *telldir*() function returns the current location associated with the named directory stream.

The *seekdir*() function sets the position of the next *readdir*() operation on the directory stream. The new position reverts to the one associated with the directory stream when the *telldir*() operation was performed.

These functions have restrictions on their use related to implementation details. Their capability can usually 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. Though this method is probably slower than using *seekdir*() and *telldir*(), there are few applications in which the capability is needed. Furthermore, directory systems that are implemented using technology such as balanced trees, where the order of presentation may vary from access to access, do not lend themselves well to any concept along these lines. For these reasons, *seekdir*() and *telldir*() are not included in POSIX.1.

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.

B.5.1.3 Set Position of Directory Stream

The *seekdir*() and *telldir*() functions were proposed for inclusion in POSIX.1, but were excluded because they are inherently unreliable when all the possible conforming implementations of the rest of POSIX.1 were considered. The problem 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 used B-trees, hashing functions, or other similar mechanisms for directory search are considered.

Even the simple goal of attempting to visit each directory entry that is unmodified between the *opendir*() and *closedir*() calls exactly once is difficult to implement reliably in the face of directory compaction and reorganization.

Since the primary need for *seekdir*() and *telldir*() is to implement file tree walks, and since such a function is likely to be included in a future revision of POSIX.1, and since in that more constrained context it appears that at least the goal of visiting unmodified nodes exactly once can be achieved, it was felt that waiting for the development of that function best served all the constituencies.

B.5.2 Working Directory

B.5.2.1 Change Current Working Directory

The *chdir*() function only affects the working directory of the current process.

The result if a **NULL** argument is passed to *chdir*() is left implementation defined because some implementations dynamically allocate space in that case.

B.5.2.2 Working Directory Pathname

Since the maximum pathname length is arbitrary unless {PATH_MAX} is defined, an application generally cannot supply a *buf* with *size* {PATH_MAX} + 1}.

Having *getcwd*() take no arguments and instead use the C function *malloc*() to produce space for the returned argument was considered. The advantage is that *getcwd*() knows how big the working directory pathname is and can allocate an appropriate amount of space. But the programmer would have to use the C function *free*() to free the resulting object, or each use of *getcwd*() would further reduce the available memory. Also, *malloc*() and *free*() are used nowhere else in POSIX.1. Finally, *getcwd*() is taken from the *SVID* {B41} , where it has the two arguments used in POSIX.1.

The older function *getwd*() was rejected for use in this context because it had only a buffer argument and no size argument, and thus had no way to prevent overwriting the buffer, except to depend on the programmer to provide a large enough buffer.

The result if a **NULL** argument is passed to *getcwd*() is left implementation defined because some implementations dynamically allocate space in that case.

If a program is operating in a directory where some (grand)parent directory does not permit reading, *getcwd*() may fail, as in most implementations it must read the directory to determine the name of the file. This can occur if search, but not read, permission is granted in an intermediate directory, or if the program is placed in that directory by some more privileged process (e.g., login). Including this error, [EACCES], makes the reporting of the error consistent and warns the application writer that *getcwd*() can fail for reasons beyond the control of the application writer or user. Some implementations can avoid this occurrence [e.g., by implementing *getcwd*() using pwd, where pwd is a set-user-root process], thus the error was made optional.

Because POSIX.1 permits the addition of other errors, this would be a common addition and yet one that applications could not be expected to deal with without this addition.

Some current implementations use {PATH_MAX}+2 bytes. These will have to be changed. Many of those same implementations also may not diagnose the [ERANGE] error properly or deal with a common bug having to do with newline in a directory name (the fix to which is essentially the same as the fix for using +1 bytes), so this is not a severe hardship.

B.5.2.3 Change Root Directory of a Process

The *chroot*() function was excluded from POSIX.1 on the basis that it was not useful to portable applications. In particular, creating an environment in which an application could run after executing a *chroot*() call is well beyond the current scope of POSIX.1.

B.5.3 General File Creation

Because there is no portable way to specify a value for the argument indicating the file mode bits (except zero), `<sys/stat.h>` is included with the functions that reference mode bits.

B.5.3.1 Open a File

Except as specified in POSIX.1, 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 B.4.2.3 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 [e.g., *creat*() on a FIFO must not remove data]. Because 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.

Implementations may deny access and return [EACCES] for reasons other than just those listed in the [EACCES] definition.

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. POSIX.1 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 7.1.1.3 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
```

See the rationale for the change from O_NDELAY to O_NONBLOCK in B.6.

B.5.3.2 Create a New File or Rewrite an Existing One

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.

B.5.3.3 Set File Creation Mask

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-specific purposes. The addition of the text that the meaning of other bits of the field are implementation defined permits these implementations to conform to POSIX.1.

B.5.3.4 Link to a File

See B.2.2.2.

Linking to a directory is restricted to the super-user in most historical implementations because this capability may produce loops in the file hierarchy or otherwise corrupt the file system. POSIX.1 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. Symbolic links are not discussed by POSIX.1. The exception for cross-file system links is intended to apply only to links that are programmatically indistinguishable from “hard” links.

B.5.4 Special File Creation

B.5.4.1 Make a Directory

See B.2.5.

The *mkdir*() function originated in 4.2BSD and was added to System V in Release 3.0.

4.3BSD detects [ENAMETOOLONG].

See B.4.2.3 about the group of a newly created directory.

B.5.4.2 Make a FIFO Special File

The syntax of this routine is intended to maintain compatibility with historical implementations of *mknod*(). The latter function was included in the *1984 /usr/group Standard* {B75} , but only for use in creating FIFO special files. The *mknod*() function was excluded from POSIX.1 as implementation defined and replaced by *mkdir*() and *mkfifo*().

See B.4.2.3 about the group of a newly created FIFO.

B.5.5 File Removal

The *rmdir*() and *rename*() functions originated in 4.2BSD, 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* {B75} was published, it contained [EEXIST] instead. When these functions were adopted into System V, the *1984 /usr/group Standard* {B75} 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.

B.5.5.1 Remove Directory Entries

Unlinking a directory is restricted to the super-user in many historical implementations for reasons given in B.5.3.4. But see B.5.5.3.

The meaning of [EBUSY] in historical implementations is “mount point busy.” Since POSIX.1 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.

B.5.5.2 Remove a Directory

See also and B.5.5.1.

B.5.5.3 Rename a File

This *rename*() function is equivalent for regular files to that defined by the C Standard {2}. Its inclusion here expands that definition to include actions on directories and specifies behavior when the *new* parameter names a file that already exists. That specification requires that the action of the function be atomic.

One of the reasons for introducing this function was to have a means of renaming directories while permitting implementations to prohibit the use of *link*() and *unlink*() with directories, thus constraining links to directories to those made by *mkdir*().

The specification that if *old* and *new* refer to the same file describes existing, although undocumented, 4.3BSD behavior. It is intended to guarantee that:

```
rename("x", "x");
```

does not remove the file.

Renaming dot or dot-dot is prohibited in order to prevent cyclical file system paths.

See also the descriptions of [ENOTEMPTY] and [ENAMETOOLONG] in B.5.5 and [EBUSY] in B.5.5.1. For a discussion of [EXDEV], see B.5.3.4.

B.5.6 File Characteristics

The *ustat*() function, which appeared in the *1984 /usr/group Standard* {B75} and is still in the *SVID* {B41} , was excluded from POSIX.1 because it is:

- — Not reliable. The amount of space available can change between the time the call is made and the time the calling process attempts to use it.
- — Not required. The only known program that uses it is the text editor ed.
- — Not readily extensible to networked systems.

B.5.6.1 Characteristics: Header and Data Structure

See B.2.5

A conforming C language application must include `<sys/stat.h>` for functions that have arguments or return values of type *mode_t*, so that symbolic values for that type can be used. An alternative would be to require that these constants are also defined by including `<sys/types.h>`.

The S_ISUID and S_ISGID bits may be cleared on any write, not just on *open*(), as some historical implementations do it.

System calls that update the time entry fields in the *stat* structure must be documented by the implementors. POSIX.1 conforming systems should not update the time entry fields for functions listed in POSIX.1 unless the standard requires that they do, except in the case of documented extensions to the standard.

Note that *st_dev* must be unique within the Local Area Network (LAN) in a “system” made up of multiple computers' file systems connected by a LAN.

Networked implementations of a POSIX.1 system must guarantee that all files visible within the file tree (including parts of the tree that may be remotely mounted from other machines on the network) on each individual processor are uniquely identified by the combination of the *st_ino* and *st_dev* fields.

B.5.6.2 Get File Status

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 will not 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.

B.5.6.3 File Accessibility

In early drafts of POSIX.1, 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 then process's real user ID is a super-user. In particular, they always return zero when testing execute permissions without regard to whether the file is executable.
2) The super-user has complete access to all files on a system. As a consequence, programs started by the super-user 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 POSIX.1 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*() was taken back out of POSIX.1.

Secure implementations will probably need an extended *access*()-like function, but there were not enough of the requirements to define it yet. This could be proposed as an extension for a future amendment to POSIX.1.

The sentence concerning appropriate privileges and execute permission bits reflects the two possibilities implemented by historical implementations when checking super-user access for X_OK.

B.5.6.4 Change File Modes

POSIX.1 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 nonexecutable 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.

B.5.6.5 Change Owner and Group of File

System III and System V allow a user to give away files; that is, the owner of a file may change its user ID to anything. This is a serious problem for implementations that are intended to meet government security regulations. Version 7 and 4.3BSD permit only the super-user to change the user ID of a file. Some government agencies (usually not ones concerned directly with security) find this limitation too confining. POSIX.1 uses “may” to permit secure implementations while not disallowing System V.

System III and System V allow the owner of a file to change the group ID to anything. Version 7 permits only the super-user to change the group ID of a file. 4.3BSD permits the owner to change the group ID of a file to its effective group ID or to any of the groups in the list of supplementary group IDs, but to no others.

Although *chown*() can be used on some systems by the file owner to change the owner and group to any desired values, the only portable use of this function is to change the group of a file to the effective GID of the calling process or to a member of its group set.

The decision to require that, for nonprivileged processes, the S_ISUID and S_ISGID bits be cleared on regular files, but only *may* be cleared on nonregular files, was to allow plans for using these bits in implementation-specified manners on directories. Similar cases could be made for other file types, so POSIX.1 does not require that these bits be cleared except on regular files. As these cases arise, the system implementors will have to determine whether these features enable any security loopholes and specify appropriate restrictions. If the implementation supports executing any file types other than regular files, the S_ISUID and S_ISGID bits should be cleared for those file types in the same way as they are on regular files.

B.5.6.6 Set File Access and Modification Times

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.

B.5.6.7 Truncate a File to a Specified Length

It was determined that *truncate*(), which takes a pathname as an argument, was of marginal additional utility in this case and was omitted in favor of the more likely use of *open*() and *ftruncate*().

B.5.7 Configurable Pathname Variables

When the run-time facility described in B.4.8 was designed, it was realized that some variables change depending on the file system. For example, it is quite feasible for a system to have two varieties of file systems mounted: a System V file system and a BSD “Fast File System.”

If limited to strictly compile-time features, no application that was widely distributed in executable binary form could rely on more than 14 bytes in a pathname component, as that is the minimum published for {NAME_MAX} in POSIX.1. The *pathconf*() function allows the application to take advantage of the most liberal file system available at run-time. In many BSD-based systems, 255 bytes are allowed for pathname components.

These values are potentially changeable at the directory level, not just at the file system. And, unlike the overall system variables, there is no guarantee that these might not change during program execution.

B.5.7.1 Get Configurable Pathname Variables

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 provided both types of file systems, an application would be forced to limit all pathname components to 14 bytes, as this would be the value specified in `<limits.h>` on such a system.

Therefore, various useful values can be queried on any pathname or file descriptor, assuming that the appropriate permissions are in place.

The value returned for the variable {PATH_MAX} indicates the longest relative pathname that could be given if the specified directory is the process’s current working directory. A process may not always be able to generate a name that long and use it if a subdirectory in the pathname crosses into a more restrictive file system.

The value returned for the variable {_POSIX_CHOWN_RESTRICTED} also applies to directories that do not have file systems mounted on them. The value may change when crossing a mount point, so applications that need to know should check for each directory. [An even easier check is to try the *chown*() function and look for an error in case it happens.]

Unlike the values returned by *sysconf*(), the pathname-oriented variables are potentially more volatile and are not guaranteed to remain constant throughout the process's lifetime. For example, in between two calls to *pathconf*(), the file system in question may have been unmounted and remounted with different characteristics.

Also note that most of the errors are optional. If one of the variables always has the same value on an implementation, the implementation need not look at *path* or *fildes* to return that value and is, therefore, not required to detect any of the errors except the meaning of [EINVAL] that indicates that the value of *name* is not valid for that variable.

If the value of any of the limits described in 2.8.4 or 2.8.5 are indeterminate (logically infinite), they will not be defined in `<limits.h>` and the *pathconf*() and *fpathconf*() functions will return –1 without changing *errno*. This can be distinguished from the case of giving an unrecognized *name* argument because *errno* will be 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.

B.6 Input and Output Primitives

System III and System V have included a flag, O_NDELAY, to mark file descriptors so that user processes would not block when doing I/O to them. If the flag is set, a *read*() or *write*() call that would otherwise need to block for data returns a value of zero instead. But a *read*() call also returns a value of zero on end-of-file, and applications have no way to distinguish between these two conditions.

BSD systems support a similar feature through a flag with the same name, but somewhat different semantics. The flag applies to all users of a file (or socket) rather than only to those sharing a file descriptor. The BSD interface provides a solution to the problem of distinguishing between a blocking condition and an end-of-file condition by returning an error, [EWOULDBLOCK], on a blocking condition.

The *1984 /usr/group Standard* {B75} includes an interface with some features from both System III/V and BSD. The overall semantics are that it applies only to a file descriptor. However, the return indication for a blocking condition is an error, [EAGAIN]. This was the starting point for POSIX.1.

The problem with the *1984 /usr/group Standard* {B75} is that it does not allow compatibility with existing applications. An implementation cannot both conform to that standard and support applications written for existing System V or BSD systems. Several changes have been considered address this issue. These include:

1) No change (from *1984 /usr/group Standard* {B75})
2) Changing to System III/V semantics
3) Changing to BSD semantics
4) Broadening POSIX.1 to allow conforming implementation a choice among these semantics
5) Changing the name of the flag from O_NDELAY
6) Changing to System III/V semantics and providing a new call to distinguish between blocking and end-of-file conditions

Alternative (5) was the consensus choice. The new name is O_NONBLOCK. This alternative allows a conforming implementation to provide backward compatibility at the source and/or object level with either System III/V or BSD systems (but POSIX.1 does not require or even suggest that this be done). It also allows a Conforming POSIX.1 Application Using Extensions the functionality to distinguish between blocking and end-of-file conditions, and to do so in as simple a manner as any of the alternatives. The greatest shortcoming was that it forces all existing System III/V and

BSD applications that use this facility to be modified in order to strictly conform to POSIX.1. This same shortcoming applies to (1) and (4) as well, and it applies to one group of applications for (2), (3), and (6).

Systems may choose to implement both O_NDELAY and O_NONBLOCK, and there is no conflict as long as an application does not turn both flags on at the same time.

See also the discussion of scope in B.6.5.1.

B.6.1 Pipes

An implementation that fails *write*() operations on *fildes*[0] or *read*()s on *fildes*[1] is not required. Historical implementations (Version 7 and System V) return the error [EBADF] in such cases. This allows implementations to set up a second pipe for full duplex operation at the same time. A conforming application that uses the *pipe*() function as described in POSIX.1 will succeed whether this second pipe is present or not.

B.6.1.1 Create an Inter-Process Channel

The wording carefully avoids using the verb “to open” in order to avoid any implication of use of *open*().

See also B.6.4.2.

B.6.2 File Descriptor Manipulation

B.6.2.1 Duplicate an Open File Descriptor

The *dup*() and *dup2*() functions are redundant. Their services are also provided by the *fcntl*() function. They have been included in POSIX.1 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 by POSIX.1 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 current draft of the Ada binding to POSIX.1.

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.

If *fildes2* is a valid file descriptor, it shall be closed, regardless of whether the function returns an indication of success or failure, unless *fildes2* is equal to *fildes*.

B.6.3 File Descriptor Deassignment

B.6.3.1 Close a File

Once a file is closed, the file descriptor no longer exists, since the integer corresponding to it no longer refers to a file.

The use of interruptible device close routines should be discouraged to avoid problems with the implicit closes of file descriptors by *exec* and *exit*(). POSIX.1 only intends to permit such behavior by specifying the [EINTR] error case.

B.6.4 Input and Output

The use of I/O with large byte counts has always presented problems. Ideas such as *lread*() and *lwrite*() (using and returning *long*s) were considered at one time. The current solution is to use abstract types on the C Standard {2} interface to *read*() and *write*() (and not to discuss common usage). The abstract types can be declared so that existing interfaces work, but can also be declared so that larger types can be represented in future implementations. It is presumed that whatever constraints limit the maximum range of *size_t* also limit portable I/O requests to the same range. POSIX.1 also limits the range further by requiring that the byte count be limited so that a signed return value remains meaningful. Since the return type is also a (signed) abstract type, the byte count can be defined by the implementation to be larger than an *int* can hold.

POSIX.1 requires that no action be taken when *nbyte* is zero. This is not intended to take precedence over detection of errors (such as invalid buffer pointers or file descriptors). This is consistent with the rest of POSIX.1, but the phrasing here could be misread to require detection of the zero case before any other errors. A value of zero is to be considered a correct value, for which the semantics are a no-op.

There were recommendations to add format parameters to *read*() and *write*() in order to handle networked transfers among heterogeneous file system and base hardware types. Such a facility may be required for support by the OSI presentation of layer services. However, it was determined that this should correspond with similar C Language facilities, and that is beyond the scope of POSIX.1. The concept was suggested to the developers of the C Standard {2} for their consideration as a possible area for future work.

In 4.3BSD, a *read*() or *write*() that is interrupted by a signal before transferring any data does not by default return an [EINTR] error, but is restarted. In 4.2BSD, 4.3BSD, and the Eighth Edition there is an additional function, *select*(), whose purpose is to pause until specified activity (data to read, space to write, etc.) is detected on specified file descriptors. It is common in applications written for those systems for *select*() to be used before *read*() in situations (such as keyboard input) where interruption of I/O due to a signal is desired. But this approach does not conform, because *select*() is not in POSIX.1. 4.3BSD semantics can be provided by extensions to POSIX.1.

POSIX.1 permits *read*() and *write*() to return the number of bytes successfully transferred when interrupted by an error. This is not simply required because it was not done by Version 7, System III, or System V, and because some hardware may not be capable of returning information about partial transfers if a device operation is interrupted. Unfortunately, this does make writing a Conforming POSIX.1 Application more difficult in circumstances where this could occur.

Requiring this behavior does not address the situation of pipelined buffers, such as might be found in streaming tape drives or other devices that read ahead of the actual requests. The signal interruption will often indicate an exceptional condition and flush all buffers. Thus, the amount read from the device may be different from the amount transferred to the application.

The issue of which files or file types are interruptible is considered an implementation design issue. This is often affected primarily by hardware and reliability issues.

There are no references to actions taken following an “unrecoverable error.” It is considered beyond the scope of POSIX.1 to describe what happens in the case of hardware errors.

B.6.4.1 Read from a File

POSIX.1 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, does modify *st_atime*.

B.6.4.2 Write to a File

An attempt to write to a pipe or FIFO has several major characteristics:

Atomic/nonatomic

A write is atomic if the whole amount written in one operation is not interleaved with data from any other process. This is useful when there are multiple writers sending data to a single reader. Applications need to know how large a write request can be expected to be performed atomically. This maximum is called {PIPE_BUF}. POSIX.1 does not say whether write requests for more than {PIPE_BUF} bytes will be atomic, but requires that writes of {PIPE_BUF} or fewer bytes shall be atomic.

Blocking/immediate

Blocking is only possible with O_NONBLOCK clear. If there is enough space for all the data requested to be written immediately, the implementation should do so. Otherwise, the process may block; that is, pause until enough space is available for writing. The effective size of a pipe or FIFO (the maximum amount that can be written in one operation without blocking) may vary dynamically, depending on the implementation, so it is not possible to specify a fixed value for it.

Complete/partial/deferred

A write request,

```
int fildes;
size_t nbyte;
ssize_t ret;
char *buf;
ret = write (fildes, buf, nbyte);
```

may return

complete: ret = *nbyte*

partial: ret < *nbyte*

This shall never happen if *nbyte* ≤ {PIPE_BUF}. If it does happen (with *nbyte* > {PIPE_BUF}), POSIX.1 does not guarantee atomicity, even if *ret* ≤ {PIPE_BUF}, because atomicity is guaranteed according 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 will never succeed. 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, POSIX.1 does not specify exactly when a partial write will be 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 will make 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 POSIX.1 requires −1 to be returned and *errno* set to [EAGAIN], most historical implementations return zero (with the O_NDELAY flag set—that flag is the historical predecessor of O_NONBLOCK, but is not itself in POSIX.1). The error indications in POSIX.1 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 (e.g., 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.

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 B.6.

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*(). Additional calls such as the common *readv*() and *writev*() would want to obey these semantics. A new "high-performance" write analog that did not follow these serialization requirements would also be permitted by this wording. POSIX.1 is also silent about any effects of application-level caching (such as that done by *stdio*).

POSIX.1 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.

POSIX.1 does not specify behavior of concurrent writes to a file from multiple processes. Applications should use some form of concurrency control.

B.6.5 Control Operations on Files

B.6.5.1 Data Definitions for File Control Operations

The main distinction between the file descriptor flags and the file status flags is scope. The former apply to a single file descriptor only, while the latter apply to all file descriptors that share a common open file description [by inheritance through *fork*() or an F_DUPFD operation with *fcntl*()]. For O_NONBLOCK, this scoping is like that of O_NDELAY in System V rather than in 4.3BSD, where the scoping for O_NDELAY is different from all the other flags accessed via the same commands.

For example:

```
        fd1 = open (pathname, oflags);
        fd2 = dup (fd1);
        fd3 = open (pathname, oflags);
```

Does an *fcntl*() call on *fd1* also apply to *fd2* or *fd3* or to both? According to POSIX.1, F_SETFD applies only to *fd1*, while F_SETFL applies to *fd1* and *fd2* but not to *fd3*. This is in agreement with all common historical implementations except for BSD with the F_SETFL command and the O_NDELAY flag (which would apply to *fd3* as well). Note that this does not force any incompatibilities in BSD implementations, because O_NDELAY is not in POSIX.1. See also .

Historically, the file descriptor flags have had only the literal values 0 and 1. POSIX.1 defines the symbolic name FD_CLOEXEC to permit a more graceful extension of this functionality. Owners of existing applications should be aware of the need to change applications using the literal values, and implementors should be aware of the existence of this practice in existing applications.

B.6.5.2 File Control

The ellipsis in the Synopsis is the syntax specified by the C Standard {2} 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 POSIX.1 are valid. It is a common error to forget this, particularly in the case of F_SETFD, because there is only one flag in POSIX.1.

POSIX.1 permits concurrent read and write access to file data using the *fcntl*() function; this is a change from the *1984 /usr/group Standard* {B75} and early POSIX.1 drafts, which included a *lockf*() function. Without concurrency controls, this feature may not be fully utilized without occasional loss of data. Since other mechanisms for creating critical regions, such as semaphores, are not included, a file record locking mechanism was thought to be appropriate. The *fcntl*() mechanism may be used to implement semaphores, although access is not first-in-first-out without extra application development effort.

Data losses occur in several ways. One is that read and write operations are not atomic, and as such a reader may get segments of new and old data if concurrently written by another process. Another occurs when several processes try to update the same record, without sequencing controls; several updates may occur in parallel and the last writer will "win." Another case is a b-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 will not handle the b-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 POSIX.1 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 of the scope of POSIX.1. Thus, an *l_sysid* member, such as found in System V, is not included in the locking structure.

Since locking is performed with *fcntl*(), rather than *lockf*(), this specification prohibits use of advisory exclusive locking on a file that is not open for writing.

Before successful return from a F_SETLK or F_SETLKW request, the previous lock type for each byte in the specified region shall be replaced by the new lock type. This can result in a previously locked region being split into smaller regions. If this would cause the number of regions being held by all processes in the system to exceed a system-imposed limit, the *fcntl*() function returns –1 with *errno* set to [ENOLCK].

Mandatory locking was a major feature of the *1984 /usr/group Standard* {B75} . 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 4.2BSD 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. Because implementation of full deadlock detection is not always feasible, the [EDEADLK] error was made optional.

B.6.5.3 Reposition Read/Write File Offset

The C Standard {2} 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 POSIX.1 that may extend the size of the file is *write*(), several C Standard {2} functions, such as *fwrite*(), *fprintf*(), etc., 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.

See B.6.5.2 for a explanation of the use of signed and unsigned offsets with *lseek*().

B.6.6 File Synchronization

The integrity of the data and files being accessed is critical to many applications. This subclause describes the facilities that allow an application to achieve the required integrity. These facilities are defined in terms of “successfully transferred,” “synchronized I/O completion,” “synchronized I/O data integrity completion,” and “synchronized I/O file integrity completion.”

Synchronized I/O is a mechanism provided to an application to ensure integrity of its data and files. A synchronized output operation provides the assurance that data that is written to an output device actually is recorded by the device. A synchronized output operation is blocking; that is, the function does not return, or an event is not posted (in the case of asynchronous I/O operations), to the application until the operation is completed. A synchronized input operation provides the assurance that data that is read from a device is actually a current image of data present on that device, any pending write operations affecting the data being read having been completed prior to returning to the requesting process.

An application specifies that synchronized I/O is to be performed on a file by specifying the O_DSYNC or O_SYNC flags on the *fcntl*() or *open*() functions. The O_DSYNC flag specifies that I/O is to be performed with respect to data integrity; that is, integrity is assured for the data being transferred. The O_SYNC flag specifies that I/O is to be performed with respect to file integrity; that is, integrity is assured not only for the data being transferred, but also for the parameters associated with the file being accessed.

In addition to the two flags just described, this section also specifies the functions *fsync*() and *fdatasyn*(), which provide for the synchronization of any system buffers and the actual peripheral device. This synchronization function shall force the completion of any pending output operations.

Models

Numerous applications require confirmation of data transfer to a nonvolatile storage medium. Probably one of the best known application areas is that of transaction processing. When data is written and if confirmation is not received, then the application is able to abort the data update and back up to the previous known state of its data, thus ensuring the integrity of its data.

Requirements

The synchronized I/O facility is meant to satisfy an application requirement for data integrity. Data integrity involves guaranteeing that data written to an output device actually reaches the output device, and, once written, that a read will obtain the data that is on the device.

1) I/O operations are synchronous, that is, they are processed in such a manner that the requesting process does not continue execution until a requested I/O operation completes.
2) It has to be possible to determine the completion of certain I/O requests in order to ensure data integrity. Information shall be provided to the process that a converse I/O operation (e.g., read after a write) issued by the same or a coexistent process will succeed.
3) Assurance shall be provided that all data written to a file is readable on a subsequent open of the file in the absence of a failure of the physical storage medium.

It is important to realize that the synchronized I/O facility does not provide a capability to recover data in the event of a device or media failure. To ensure that critical data can be recovered, techniques such as “mirroring” (maintaining duplicate copies of file systems) and “file duplexing” (maintaining duplicate copies of files, preferably in different file systems) should be used. These techniques are not addressed at this time since they are out of scope of the facilities currently being addressed in this standard.

Standardization Issues

In defining “successfully transferred,” it is not the intention of this standard to specify the hardware characteristics of the storage device. The purpose of synchronized I/O is to ensure that application data is an “image” of what resides on “nonvolatile storage.” A buffer cache, unless it is secure from power failure, does not qualify as “nonvolatile storage.“

In regards to the references to “system buffers,” it is not the intent of this standard to require a buffered implementation. It is the intent of this standard that if a buffered implementation is used, a mechanism is provided to maintain consistency, and hence integrity, between the buffers and the actual device.

It may not always be possible to tell if a satisfied read request reflects what is physically on the medium. All that can be guaranteed from the standpoint of an operating system is that the physical operation has been initiated and that the device has acknowledged receipt of the information. Caching disk controllers make the guarantee of data residency difficult.

Separate Synchronized I/O Data and File Integrity

The difference between synchronized I/O data integrity and synchronized I/O file integrity is subtle, but important to understand. Synchronized I/O data integrity ensures that the data and all information required to subsequently retrieve the data has been successfully transferred. Synchronized I/O file integrity is a superset of data integrity. File integrity requires data integrity, but extends the concept one step further. File integrity not only guarantees the integrity of the data, but also ensures that all other file system information relevant to that data is successfully transferred. This includes related file system information not required for actual data access, e.g., time of last update, time of last access, etc.

At first thought one might assume that the single concept of file integrity would be adequate and that data integrity could be eliminated. However, possible performance implications of conforming implementations make support of both concepts desirable. For example, it is quite possible that support of file integrity will require extra I/O operations to update the nonessential file system information. As another example, many systems have an access time that is updated on every read or write; this information may not be worth the performance cost for some time-critical realtime applications. Thus, the performance cost of synchronized I/O file integrity could very likely be much greater than that of just synchronized I/O data integrity. Forcing all users of synchronized I/O to bear this possibly undesired and unnecessary cost was deemed unacceptable. Therefore, the separate features of synchronized I/O data integrity and synchronized I/O file integrity are both supported.

Rationale for New Interface

The synchronized I/O interface described in this standard describes an interface for two levels of integrity, and the ability to request completion of outstanding write operations before completion of read operations. The O_SYNC flag described here is essentially identical to the O_SYNC of existing practice. The O_DSYNC specifies a less stringent form of integrity. It requires only that the data and the information to access it be done in a synchronized manner; the file status information may be updated in a more “leisurely” manner.

Since existing practice does not impose synchronized I/O on input operations, this standard extends this concept to include input operations via the O_RSYNC flag.

The *fsync*() function provides that any pending output be completed in a manner as if the O_SYNC flag had been set. The *fdatasync*() function was added to provide an similar capability, but as if the O_DSYNC flag had been set.

Technical Considerations

The following issues arise with respect to implementation or use of synchronized I/O.

- *FIFO flies or pipes*
  The purpose of synchronized I/O is to give assurance to an application that data that is written to a storage device actually is present on the physical media of the device. Since FIFO files or pipes may be implemented without access to any physical storage media, the concept of synchronized I/O does not apply. If an application desires to pass data in a physically secure manner, i.e., immune to system crashes, the data should be passed via a file with synchronized I/O. The standard is silent on the implementation of synchronized I/O to pipes. This will allow a particular implementation to extend the concept of data integrity via synchronized I/O to pipes.
- *Synchronized read*
  A synchronized read “provides assurance that data that is read from a device is actually a current image of data present on that device”. This image may be obtained by actually reading the data from the storage device, or it may be obtained from a buffer, if the buffer data is identical to, i.e., is an image of, the data on the storage device. The standard does not specify how the implementation obtains the data; this is left to the implementor to determine.
- *File system considerations*
  It should be noted that not all file systems may, or even need to, support synchronized I/O. Consequently, when synchronized I/O is specified (the O_SYNC or O_DSYNC flags are specified) on the *open*() or *fcntl*() functions, the function may fail ([EINVAL] returned) due to the file system not being able to support synchronized I/O for the specified file.
- *Raw I/0*
  The operating system cannot protect users if they circumvent the provided protection mechanisms by using raw I/O. Consequently, the standard is silent on the issue of raw I/O. It is assumed that if an application uses raw I/O, then the application is providing its own mechanisms for ensuring data and file integrity.

B.6.6.1 Synchronize the State of a File

The *fsync*() function has been adapted from IEEE P1003.1a , with its functionality described in its entirety in this standard. This “base” function is supported if the symbol {_POSIX_FSYNC} is defined. The “base” function is defined quite loosely to meet the requirements of nonrealtime applications. This loosely defined *fsync*() function is deemed by the standard developers to be unacceptable for realtime applications. Consequently, if the implementation supports the Synchronized Input and Output option, then a more rigorous specification of the function applies. For example, in the less rigorous case, a null implementation is permitted. In the more rigorous case, a null implementation would not be acceptable. Note that if {_POSIX_SYNCHRONIZED_IO} is defined, then the *fsync*() function is defined and the more rigorous specification applies, regardless of whether {_POSIX_FSYNC} is defined or not. The "base" function is separated out from the rest of the functions described in this section so that it may be specifically called out by an Applications Environment Profile (AEP), for example, the POSIX Platform AEP (IEEE P1003.18).

The *fsync*() function should be used by programs that require a file to be in a known state; for example, a program that contains a simple transaction facility might use it to ensure that all modifications to a file or files caused by a transaction are recorded.

The *fsync*() function is intended to force a physical write of data from the buffer cache, and to assure that after a system crash or other failure that all data up to the time of the *fsync*() call is recorded on the disk. Since the concepts of "buffer cache," "system crash," "physical write," and "nonvolatile storage" are not defined here, the wording has to be more abstract.

In the less rigorous case, the wording relies heavily on the conformance document to tell the user what can be expected from the system. It is explicitly intended that a null implementation is permitted. This could be valid in the case where the system cannot assure nonvolatile storage under any circumstances or when the system is highly fault-tolerant and the functionality is not required. In the middle ground between these extremes, *fsync*() might or might not actually cause data to be written where it is safe from a power failure. The conformance document should identify at least that one configuration exists (and how to obtain that configuration) where this can be assured for at least some files that the user can select to use for critical data. It is not intended that an exhaustive list is required, but rather sufficient information is provided to let the user determine that if he or she has critical data he or she can configure her system to allow it to be written to nonvolatile 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.

B.6.6.2 Synchronize the Data of a File

There is no specific rationale for this subclause.

B.6.7 Asynchronous Input and Output

Many applications need to interact with the I/O subsystem in an asynchronous manner. The asynchronous I/O mechanism provides the ability to overlap application processing and I/O operations initiated by the application. The asynchronous I/O mechanism allows a single process to perform I/O simultaneously to a single file multiple times or to multiple files multiple times.

Overview

Asynchronous I/O operations proceed in logical parallel with the processing done by the application after the asynchronous I/O has been initiated. Other than this difference, asynchronous I/O behaves similarly to normal I/O using *read*(), *write*(), *lseek*(), and *fsync*(). The effect of issuing an asynchronous I/O request is as if a separate thread of execution were to perform atomically the implied *lseek*() operation, if any, and then the requested I/O operation [either *read*(), *write*(), or *fsync*()]. There is no seek implied with a call to *aio_fsync*(). Concurrent asynchronous operations and synchronous operations applied to the same file update the file as if the I/O operations had proceeded serially.

When asynchronous I/O completes, a signal can be delivered to the application to indicate the completion of the I/O. This signal can be used to indicate that buffers and control blocks used for asynchronous I/O can be re-used. Signal delivery is not required for an asynchronous operation and may be turned off on a per-operation basis by the application. Signals may also be synchronously polled using *aio_suspend*(), *sigtimedwait*(), or *sigwaitinfo*().

Normal I/O has a return value and an error status associated with it. Asynchronous I/O returns a value and an error status when the operation is first submitted, but that only relates to whether the operation was successfully queued up for servicing. The I/O operation itself also has a return status and an error value. To allow the application to retrieve the return status and the error value, functions are provided that, given the address of an asynchronous I/O control block, yield the return and error status associated with the operation. Until an asynchronous I/O operation is done, its error status shall be [EINPROGRESS]. Thus, an application can poll for completion of an asynchronous I/O operation by waiting for the error status to become equal to a value other than [EINPROGRESS]. The return status of an asynchronous I/O operation is undefined so long as the error status is equal to [EINPROGRESS].

Storage for asynchronous operation return and error status may be limited. Submission of asynchronous I/O operations may fail if this storage is exceeded. When an application retrieves the return status of a given asynchronous operation, therefore, any system-maintained storage used for this status and the error status may be reclaimed for use by other asynchronous operations.

Asynchronous I/O can be performed on file descriptors that have been enabled for POSIX.1b *synchronized I/O*. In this case, the I/O operation still occurs asynchronously, as defined herein; however, the asynchronous operation I/O in this case is not completed until the I/O has reached either the state of synchronized I/O data integrity completion or synchronized I/O file integrity completion, depending on the sort of synchronized I/O that is enabled on the file descriptor.

Models

Three models illustrate the use of asynchronous I/O: a journalization model, a data acquisition model, and a model of the use of asynchronous I/O in supercomputing applications.

— *Journalization model*
  Many realtime applications perform low-priority journalizing functions. Journalizing requires that logging records be queued for output without blocking the initiating process.
— *Data acquisition model*
  A data acquisition process may also serve as a model. The process has two or more channels delivering intermittent data that must be read within a certain time. The process issues one asynchronous read on each channel. When one of the channels needs data collection, the process reads the data and posts it through an asynchronous write to secondary memory for future processing.
— *Supercomputing model*
  The supercomputing community has used asynchronous I/O much like that specified herein for many years. This community requires the ability to perform multiple I/O operations to multiple devices with a minimal number of entries to “the system”; each entry to “the system” provokes a major delay in operations when compared to the normal progress made by the application. This existing practice motivated the use of combined *lseek*() and *read*() or *write*() calls, as well as the *lio_listio*() call. Another common practice is to disable signal notification for I/O completion, and simply poll for I/O completion at some interval by which the I/O should be completed. Likewise, interfaces like *aio_cancel*() have been in successful commercial use for many years. Note also that an underlying implementation of asynchronous I/O will require the ability, at least internally, to cancel outstanding asynchronous I/O, at least when the process exits (Consider an asynchronous read from a terminal, when the process intends to exit immediately).

Requirements

Asynchronous input and output for realtime implementations have these requirements:

- The ability to queue multiple asynchronous read and write operations to a single open instance. Both sequential and random access should be supported.
- The ability to queue asynchronous read and write operations to multiple open instances.
- The ability to obtain completion status information by polling and/or asynchronous event notification.
- Asynchronous event notification on asynchronous I/O completion is optional.
- It has to be possible for the application to associate the event with the *aiocbp* for the operation that generated the event.
- The ability to cancel queued requests.
- The ability to wait upon asynchronous I/O completion in conjunction with other types of events.
- The ability to accept an *aio_read*() and an *aio_cancel*() for a device that accepts a *read*(), and the ability to accept an *aio_write*() and an *aio_cancel*() for a device that accepts a *write*(). This does not imply that the operation is asynchronous.

Standardization Issues

The following issues are addressed by the standardization of asynchronous I/O.

- *Rationale for new interface*
  Nonblocking I/O does not satisfy the needs of either realtime or high-performance computing models; these models require that a process overlap program execution and I/O processing. Realtime applications will often make use of direct I/O to or from the address space of the process, or require synchronized (unbuffered) I/O; they also require the ability to overlap this I/O with other computation. In addition, asynchronous I/O allows an application to keep a device busy at all times, possibly achieving greater throughput. Supercomputing and database architectures will often have specialized hardware that can provide true asynchrony underlying the logical asynchrony provided by this interface. In addition, asynchronous I/O should be supported by all types of files and devices in the same manner.
- *Effect of buffering*
  If asynchronous I/O is performed on a file that is buffered prior to being actually written to the device, it is possible that asynchronous I/O will offer no performance advantage over normal I/O; the cycles "stolen" to perform the asynchronous I/O will be taken away from the running process and the I/O will occur at interrupt time. This potential lack of gain in performance in no way obviates the need for asynchronous I/O by realtime applications, which very often will use specialized hardware support; multiple processors; and/or unbuffered, synchronized I/O.

B.6.7.1 Data Definitions for Asynchronous Input and Output

This revision makes consistent the semantics of the members of the *sigevent* structure, particularly in the definitions of *lio_listio*() and *aio_fsync*(). The requirements previously listed are subsumed by the reference to 3.3.1.2.

B.6.7.2 Asynchronous Read

There is no specific rationale for this subclause.

B.6.7.3 Asynchronous Write

There is no specific rationale for this subclause.

B.6.7.4 List Directed I/O

This revision makes consistent the semantics of the members of the *sigevent* structure.

Although it may appear that there are inconsistencies in the specified circumstances for error codes, the error code [EIO] applies when any circumstance relating to an individual operation makes that operation fail. This might be due to a badly formulated request (e.g., the *aio_lio_opcode* field is invalid, and *aio_error*() returns [EINVAL]) or might arise from application behavior (e.g., 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.

B.6.7.5 Retrieve Error Status of Asynchronous I/O Operation

There is no specific rationale for this subclause.

B.6.7.6 Retrieve Return Status of Asynchronous I/O Operation

There is no specific rationale for this subclause.

B.6.7.7 Cancel Asynchronous I/O Request

There is no specific rationale for this subclause.

B.6.7.8 Wait for Asynchronous I/O Request

There is no specific rationale for this subclause.

B.6.7.9 Asynchronous File Synchronization

This revision makes consistent the semantics of the members of the *sigevent* structure.

B.7 Device- and Class-Specific Functions

There were several sources of difficulties involved with using historical interfaces as the basis of this section:

1) The basic Version 7 *ioctl*() mechanism is difficult to specify adequately, due to its use of a third argument that varies in both size and type according to the second, command, argument.
2) System III introduced and System V continued *ioctl*() commands that are completely different from those of Version 7.
3) 4.2BSD and other BSD systems added to the basic Version 7 *ioctl*() command set; some of these were for features such as job control that POSIX.1 eventually adopted.
4) None of the basic historical implementations are adequate in an international environment. This concern is not technically within the scope of POSIX.1, but the goal of POSIX.1 was to mandate no unnecessary impediments to internationalization.

The *1984 /usr/group Standard* {B75} attempted to specify a portable mechanism that application writers could use to get and set the modes of an asynchronous terminal. The intention of that committee was to provide an interface that was neither implementation specific nor hardware dependent. Initial proposals dealt with high-level routines similar to the *curses* library (available on most historical implementations). In such an implementation, the user interface would consist of calls similar to:

```
setraw();
setcooked();
```

It was quickly pointed out that if such routines were standardized, the definition of “raw” and “cooked” would have to be provided. If these modes were not well defined in POSIX.1, application code could not be written in a portable way. However, the definition of the terms would force low-level concepts to be included in a supposedly high-level interface definition.

Focus was given to the necessary low-level attributes that were needed to support the necessary terminal characteristics (e.g., line speeds, raw mode, cooked mode, etc.). After considerable debate, a structure similar to, but more flexible than, the System III *termio* was accepted. The format of that structure, referred to as the *termios* structure, has formed the basis for the current section.

A method was needed to communicate with the system about the *termios* information. Proposals included:

1) The *ioctl*() function as in System V. This had the same problems as mentioned previously for the Version 7 *ioctl*() function and was basically identical to it. Another problem was that the direction of the command (whether information is written from or read into the third argument) was not specified—in historical implementations, only the device driver knows this information. This was a problem for networked implementations. It was also a problem that there was no size parameter to specify the variable size of the third argument, and there was a similar problem with its type.
2) An *iocntl*() function with additional arguments specifying direction, type, and size. But these new arguments did not help application writers, who would have no control over their values, which would have to match each command exactly. The new arguments did, however, solve the problems of networked implementations. And *iocntl*() would have been implementable in terms of *ioctl*() on historical implementations (without need for modifying existing code), although it would have been easy to update existing code to use the arguments directly.
3) A termcntl function with the same arguments as proposed for the *iocntl*() function. The difference was that *termcntl*() would be limited to terminal interface functions; there would be other interface functions, such as a *tapecntl*() function for tape interfaces, rather than a single general device interface routine.
4) Unspecified functions. The issue of what the interface function(s) should be called was avoided for many of the early drafts while details of the information to be handled was of prime concern. The resulting specification resembled the information in System V, but attempted to avoid problems of case, speed, networks, and internationalization.

Specific *tc**() functions[16] to replace each *ioctl*() function were finally incorporated into POSIX.1, instead of any of the previously mentioned proposals.

The issue of modem control was excluded from POSIX.1 on the grounds that

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

B.7.1 General Terminal Interface

If the implementation does not support this interface on any device types, it should behave as if it were being used on a device that is not a terminal device (in most cases *errno* will be set to [ENOTTY]) on return from functions defined by this interface. This is based on the fact that many applications are written to run both interactively and in some noninteractive mode, and they adapt themselves at run time. Requiring that they all be modified to test an environment variable to determine if they should try to adapt is unnecessary. On a system that provides no Section 7 interface, providing all the entry points as stubs that return [ENOTTY] (or an equivalent, as appropriate) has the same effect and requires no changes to the application.

Although the needs of both interface implementors and application developers were addressed throughout POSIX.1, this section pays more attention to the needs of the latter. This is because, while many aspects of the programming interface can be hidden from the user by the application developer, the terminal interface is usually a large part of the user interface. Although to some extent the application developer can build missing features or work around

[16]The notation *tc**() is reminiscent of shell pattern matching notation and is an abbreviated way of referring to all functions beginning with the letters “tc.”

inappropriate ones, the difficulties of doing that are greater in the terminal interface than elsewhere. For example, efficiency prohibits the average program from interpreting every character passing through it in order to simulate character erase, line kill, etc. These functions should usually be done by the operating system, possibly at the interrupt level.

The *tc**() functions were introduced as a way of avoiding the problems inherent in the traditional *ioctl*() function and in variants of it that were proposed. For example, *tcsetattr*() is specified in place of the use of the TCSETA *ioctl*() command function. This allows specification of all the arguments in a manner consistent with the C Standard {2}, unlike the varying third argument of *ioctl*(), which is sometimes a pointer (to any of many different types) and sometimes an *int*.

The advantages of this new method include:

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

The disadvantages include:

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

B.7.1.1 Interface Characteristics

B.7.1.1.1 Opening a Terminal Device File

Further implications of the effects of CLOCAL are discussed in 7.1.2.4.

B.7.1.1.2 Process Groups

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

The cases where a controlling terminal has no foreground process group occur when all processes in the foreground process group either terminate and are waited for or join other process groups via *setpgid*() or *setsid*(). If the process group leader terminates, this is the first case described; if it leaves the process group via *setpgid*(), this is the second case described [a process group leader cannot successfully call *setsid*()]. When one of those cases causes a controlling terminal to have no foreground process group, it has two visible effects on applications. The first is the value returned by *tcgetpgrp*(), as discussed in 7.2.3 and B.7.2.3. The second (which occurs. only in the case where the process group leader terminates) is the sending of signals in response to special input characters. The intent of POSIX.1 is that no process group be wrongly identified as the foreground process group by *tcgetpgrp*() or unintentionally receive signals because of placement into the foreground.

In 4.3BSD, the old process group ID continues to be used to identify the foreground process group and is returned by the function equivalent to *tcgetpgrp*(). In that implementation it is possible for a newly created process to be assigned the same value as a process ID and then form a new process group with the same value as a process group ID. The result is that the new process group would receive signals from this terminal for no apparent reason, and POSIX.1 precludes this by forbidding a process group from entering the foreground in this way. It would be more direct to place part of the requirement made by the last sentence under 3.1.1, but there is no convenient way for that subclause to refer to the value that *tcgetpgrp*() returns, since in this case there is no process group and thus no process group ID.

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

Another possibility is to recognize when the last process stops using the terminal's foreground process group ID, which is when the process group lifetime ends, and to change the terminal's foreground process group ID to a reserved value that is never used as a process ID or process group ID. (See the definition of process group lifetime in 2.2.2.) The process ID can then be reserved until the terminal has another foreground process group.

The 4.3BSD implementation permits the leader (and only member) of the foreground process group to leave the process group by calling the equivalent of *setpgid*() and to later return, expecting to return to the foreground. There are no known application needs for this behavior, and POSIX.1 neither requires nor forbids it (except that it is forbidden for session leaders) by leaving it unspecified.

B.7.1.1.3 The Controlling Terminal

POSIX.1 does not specify a mechanism by which to allocate a controlling terminal. This is normally done by a system utility (such as `getty`) and is considered an administrative feature outside the scope of POSIX.1.

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

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

B.7.1.1.4 Terminal Access Control

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

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

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

The terminal driver repeatedly performs the foreground/background/orphaned process group checks whenever a process is about to access the terminal. In the case of *write*() or the control functions in 7.2, the check is performed at the entry of the function. In the case of *read*(), the check is performed not only at the entry of the function, but also after blocking the process to wait for input characters (if necessary). That is, once the driver has determined that the process calling the *read*() function is in the foreground, it attempts to retrieve characters from the input queue. If the queue is empty, it blocks the process waiting for characters. When characters are available and control is returned to the driver, the terminal driver must return to the repeated foreground/background/orphaned process group check again. The process may have moved from the foreground to the background while it was blocked waiting for input characters.

B.7.1.1.5 Input Processing and Reading Data

There is no additional rationale provided for this subclause.

B.7.1.1.6 Canonical Mode Input Processing

The term “character” is intended here. ERASE should erase the last character, not the last byte. In the case of multibyte characters, these two may be different. 4.3BSD has a WERASE character that erases the last “word” typed (but not any preceding blanks or tabs). A word is defined as a sequence of nonblank characters, with tabs counted as blanks. Like ERASE, WERASE does not erase beyond the beginning of the line. This WERASE feature has not been specified in POSIX.1 because it is difficult to define in the international environment. It is only useful for languages where words are delimited by blanks. In some ideographic languages, such as Japanese and Chinese, words are not delimited at all. The WERASE character should presumably take one back to the beginning of a sentence in those cases; practically, this means it would not get much use for those languages.

It should be noted that there is a possible inherent deadlock if the application and implementation conflict on the value of MAX_CANON. With ICANON set (if IXOFF is enabled) and more than MAX_CANON characters transmitted without a linefeed, transmission will be stopped, the linefeed (or carriage return when ICRLF is set) will never arrive, and the *read*() will never be satisfied.

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

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

B.7.1.1.7 Noncanonical Mode Input Processing

Some points to note about MIN and TIME:

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

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

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

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

B.7.1.1.8 Writing Data and Output Processing

There is no additional rationale provided for this subclause.

B.7.1.1.9 Special Characters

There is no additional rationale provided for this subclause.

B.7.1.1.10 Modem Disconnect

There is no additional rationale provided for this subclause.

B.7.1.1.11 Closing a Terminal Device File

POSIX.1 is silent on whether a *close*() will block on waiting for transmission to drain, or even if a *close*() might cause a flush of pending output. If the application is concerned about this, it should call the appropriate function, such as *tcdrain*(), to ensure the desired behavior.

B.7.1.2 Parameters That Can Be Set

B.7.1.2.1 *termios* Structure

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

B.7.1.2.2 Input Modes

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

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

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

In dealing with multibyte characters, the consequences of a parity error in such a character, or in an escape sequence affecting the current character set, are beyond the scope of POSIX.1 and are best dealt with by the application processing the multibyte characters.

B.7.1.2.3 Output Modes

POSIX.1 does not describe postprocessing of output to a terminal or detailed control of that from a portable application. (That is, translation of newline to carriage return followed by linefeed or tab processing.) There is nothing that a portable application should do to its output for a terminal because that would require knowledge of the operation of the terminal. It is the responsibility of the operating system to provide postprocessing appropriate to the output device, whether it is a terminal or some other type of device.

Extensions to POSIX.1 to control the type of postprocessing already exist and are expected to continue into the future. The control of these features is primarily to adjust the interface between the system and the terminal device so the output appears on the display correctly. This should be set up before use by any application.

In general, both the input and output modes should not be set absolutely, but rather modified from the inherited state.

B.7.1.2.4 Control Modes

This subclause could be misread that the symbol "CSIZE" is a title in Table 7.3. Although it does serve that function, it is also a required symbol, as a literal reading of POSIX.1 (and the caveats about typography) would indicate.

B.7.1.2.5 Local Modes

Noncanonical mode is provided to allow fast bursts of input to be read efficiently while still allowing single-character input.

The ECHONL function historically has been in many implementations. Since there seems to be no technical problem with supporting ECHONL, it is included in POSIX.1 to increase consensus.

The alternate behavior possible when ECHOK or ECHOE are specified with ICANON is permitted as a compromise depending on what the actual terminal hardware can do. Erasing characters and lines is preferred, but is not always possible.

B.7.1.2.6 Special Control Characters

Permitting VMIN and VTIME to overlap with VEOF and VEOL was a compromise for historical implementations. Only when backwards compatibility of object code is a serious concern to an implementor should an implementation continue this practice. Correct applications that work with the overlap (at the source level) should also work if it is not present, but not the reverse.

B.7.1.2.7 Baud Rate Values

There is no additional rationale provided for this subclause.

B.7.1.3 Baud Rate Functions

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.

These functions do not take numbers as arguments, but rather symbolic names. There are two reasons for this:

- Historically, numbers were not used because of the way the rate was stored in the data structure. This is retained even though an interface function is now used.
- 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 to accept, as an extension, a number (such as 126) if it wished, and because the encoding of the Bxxx symbols is not specified, this can be done so no ambiguity is introduced.

Setting the input baud rate to zero was a mechanism to allow for split baud rates. Clarifications to this version of POSIX.1 have made it possible to determine if 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. POSIX.1 does not preclude B0 from being defined as the value 0; in fact, implementations will likely benefit from the two being equivalent. POSIX.1 does not fully specify whether the previous *cfsetispeed*() value is retained after a *tcgetattr*() as the actual value or as zero. Therefore, portable 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 nonportable 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.

B.7.2 General Terminal Interface Control Functions

The restrictions described in this subclause on access from processes in background process groups controls apply only to a process that is accessing its controlling terminal. (See B.7.1.1.4).

Care must be taken when changing the terminal attributes. Applications should always do a *tcgetattr*(), save the *termios* structure values returned, and then do a *tcsetattr*() changing only the necessary fields. The application should use the values saved from the *tcgetattr*() to reset the terminal state whenever it is done with the terminal. This is necessary because terminal attributes apply to the underlying port and not to each individual open instance; that is, all processes that have used the terminal see the latest attribute changes.

A program that uses these functions should be written to catch all signals and take other appropriate actions to assure that when the program terminates, whether planned or not, the terminal device's state is restored to its original state. See also B.7.1.

Existing practice dealing with error returns when only part of a request can be honored is based on calls to the *ioctl*() function. In historical BSD and System V implementations, the corresponding *ioctl*() returns zero if the requested actions were semantically correct, even if some of the requested changes could not be made. Many existing applications assume this behavior and would no longer work correctly if the return value were changed from zero to –1 in this case.

Note that either specification has a problem. When zero is returned, it implies everything succeeded even if some of the changes were not made. When –1 is returned, it implies everything failed even though some of the changes were made.

Applications that need all of the requested changes made to work properly should follow *tcsetattr*() with a call to *tcgetattr*() and compare the appropriate field values.

B.7.2.1 Get and Set State

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.

B.7.2.2 Line Control Functions

There is no additional rationale provided for this subclause.

B.7.2.3 Get Foreground Process Group ID

The *tcgetpgrp*() function has identical functionality to the 4.2BSD *ioctl*() function TIOCGPGRP except for the additional security restriction that the referenced terminal must be the controlling terminal for the calling process.

In the case where there is no foreground process group, returning an error rather than a positive value was considered. This was rejected because existing applications based on either IEEE Std 1003.1-1988 or 4.3BSD are likely to consider errors from this call or the BSD equivalent to be catastrophic and respond inappropriately. Such applications implicitly assume that this case does not exist, and the positive return value is the only solution that permits them to behave properly even when they do encounter it. No application has been identified that can benefit from distinguishing between this case and the case of a valid foreground process group other than its own. Therefore, requiring or permitting any other solution would cause more application portability problems with no corresponding benefit to applications. The value must be positive, not zero, because applications may use the negation as the *pid* argument to *kill*(). In addition, the value 1 must not be used so that an attempt to send a signal to this (nonexistent) process group does not result in broadcasting a signal unintentionally. See also B.7.1.1.2.

B.7.2.4 Set Foreground Process Group ID

The *tcsetpgrp*() function has identical functionality to the 4.2BSD *ioctl*() function TIOCSPGRP except for the additional security restrictions that the referenced terminal must be the controlling terminal for the calling process and the specified new process group must be currently in use in the caller's session.

B.8 Language-Specific Services for the C Programming Language

See the discussion of C functions in B.1.1.1.

Common usage may be defined by historical publications such as *The C Programming Language* {B57} .

The null set of supported languages is allowed.

The list of functions comprises the list of “common-usage” functions and also includes those that are not in common usage that are addressed by POSIX.1. The rules for common-usage conformance to POSIX.1 address whether the functions that are not generally considered in common usage are implemented. There are a large number of functions found in various systems that, although frequently found, are not broadly enough available to be considered in common usage. The *signal*() function (although in common usage) is omitted because applications conforming to POSIX.1 should use the more reliable *sigaction*() interface instead.

B.8.1 Referenced C Language Routines

The *raise*() function could logically either direct the signal to the calling thread or to the calling process. Directing it to the thread matches both the usual existing implementations and the C Standard {2} rationale.

B.8.1.1 Extensions to Time Functions

System V uses the **TZ** environment variable to set some information about time. It has the form (spaces inserted for clarity):

```
        std offset dst
```

where the first three characters (*std*) are the name of the standard time zone, the digits that follow (*offset*) represent the time added to the local time zone to arrive at Coordinated Universal Time, and the next three characters (*dst*) are the name of the summer time zone. The meaning of *offset* implies that most sites west of the Prime Meridian will have a positive offset (preceded by an optional plus sign, “+”), while most sites east of the Prime Meridian will have a negative offset (preceded by a minus sign, “–”). Both *std* and *offset* are required; if *dst* is missing, summer time does not apply.

Currently, the UNIX system *localtime*() function translates a number of seconds since the Epoch into a detailed breakdown of that time. This breakdown includes

1) Time of day: Hours, minutes, and seconds.
2) Day of the month, month of the year, and the year.
3) Day of the week and day of the year (Julian day).

4) Whether or not summer (daylight saving) time is in effect.

It is the first and last items that present a problem: the time of the day depends on whether or not summer time is in effect. Whether or not summer time is in effect depends on the locale and date.

Most historical systems had time-zone rules compiled into the C library. These rules usually represented United States rules for 1970 to 1986. This did not accommodate the changes of 1987, nor other world variations ($^1/_2$-hour time, double daylight time, and solar time being common, but not complete, examples). Some recent systems addressed these problems in various ways.

Having the rules compiled into the program made binary distributions that accommodated all the variations (including sudden changes to the law), and per-process rule changes, difficult at best.

POSIX.1 includes a way of specifying the time zone in the **TZ** string, but only permits one time-zone pattern at a time, thus not dealing with different patterns in previous years and with such issues as solar time. Methods exist to deal with all the problems above. The method in POSIX.1 appears to be simpler to implement and may be faster in execution when it is adequate. POSIX.1 also permits an implementation-defined rule set that begins with a colon. (The previous format cannot begin with a colon.)

Rules of the form AAAn or AAAnBBB (the style used in many historical implementations) do not carry with them any statement about the start and end of daylight time (neither the date nor the time of day; the default to 02:00 not applying if no *rule* is present at all), thus implying that the implementation must provide the appropriate *rule*s. An implementation may provide those rules in any way it sees fit, as long as the constraints implied by the **TZ** string as provided by the user are met. Specifically, the implementation may use the string as an index into a table, which may reside either on disk or in memory. Such tables could contain rules that are sensitive to the year to which they are applied, again since the user did not specify the exact *rule*. (Although impractical, every possible **TZ** string could be represented in a table, as a detail of implementation; the less specific the user is about the **TZ** string, the more freedom the implementation has to interpret it.)

There is at least one public-domain time-zone implementation (the Olson/Harris method) that uses nonspecific **TZ** strings and a table, as described previously, and handles all the general time-zone problems mentioned above. This implementation also appears in a late release of 4.3BSD. If this implementation honors all the specifications provided in the **TZ** string, it would conform to POSIX.1. Nothing precludes the implementation from adding information beyond that given by the user in the **TZ** string.

The fully specified **TZ** environment variable extends the historical meaning to also include a rule for when to use standard time and when to use summer time. Southern hemisphere time zones are supported by allowing the first *rule date* (change to summer time) to be later in the year than the second *rule date* (change to standard time).

This mechanism accommodates the “floating day” rules (for example “last Sunday in October”) used in the United States and Canada (and the European Economic Community for the last several years). In theory, **TZ** only has to be set once and then never touched again unless the law is changed.

Julian dates are proposed with two syntaxes, one zero-based, the other one-based. They are here for historical reasons. The one-based counting (*J*) is used more commonly in Europe (and on calendars people may use for reference). The zero-based counting (*n*) is used currently in some implementations and should be kept for historical reasons as well as being the only way to specify Leap Day.

It is expected that the leading colon format will allow systems to implement an even broader range of specifications for the time zone without having to resort to a file or permit naming an explicit file containing the appropriate rules.

The specification in POSIX.1 for **TZ** assumes that very few programs need to be historically accurate as long as the relative timing of two events is preserved.

Summer time is governed by both locale and date. This proposal only handles the locale dependency. Using an implementation-defined file format for either the entire **TZ** variable or to specify the *rules* for a particular time zone is allowed as a means by which both the locale and date dependency can be handled.

Since historical implementations do not examine **TZ** beyond the assumed end of *dst*, it is possible literally to extend **TZ** and break very little existing software. Since much historical software does not function outside the US time zones, minor changes to **TZ** (such as extending *offset* to be *hh*:*mm*—as long as the colon and minutes, *:mm*, are optional) should have little effect.

POSIX.1 is intentionally silent about values of **TZ** that do not fit either of the specified forms. It simply requires that **TZ** values that follow those forms be interpreted as specified.

B.8.1.2 Extensions to *setlocale*() Function

The C Standard {2} defines a collection of interfaces to support internationalization. One of the most significant aspects of these interfaces 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) String Handling (i.e., collating)
3) Date/Time Formatting
4) Numeric Editing

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 the following:

```
char     *setlocale (int category, const char *locale);
```

where *category* is the name of one of five categories, namely

```
LC_CTYPE
LC_COLLATE
LC_TIME
LC_MONETARY
LC_NUMERIC
```

In addition, a special value, called LC_ALL, directs *setlocale*() to set all categories.

The *locale* argument is a character string that points to a specific setting for the international environment, or locale. There are three preset values for the locale argument, namely

`"C"`	Specifies the minimal environment for C translation. If *setlocale*() is not invoked, the "C" locale is the default.
`"POSIX"`	Specifies a locale that is the same as `"C"` for the attributes defined by the C Standard {2} and POSIX.1, but may contain extensions. The wording permits extensions by standards, specifically that of ISO/IEC 9945-2 {B36} , which is expected to use the same symbol, and by future versions of POSIX.1.
`""`	Specifies an implementation-defined native environment.
NULL	Used to direct *setlocale*() to query the current international environment and return the name of the locale.

This subclause describes the behavior of an implementation of *setlocale*() and its use of environment variables in controlling this behavior on POSIX.1-based systems. 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 following subclauses describe the behavior of *setlocale*() in these two areas. Since it is difficult to describe the behavior in words, examples will be 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:

$$\mathit{setlocale}\,(\left\{\begin{array}{l} \mathit{LC\_ALL} \\ \mathit{LC\_CTYPE} \\ \mathit{LC\_COLLATE} \\ \mathit{LC\_TIME} \\ \mathit{LC\_NUMERIC} \\ \mathit{LC\_MONETARY} \end{array}\right\}, (\mathit{char}\ *)\,\mathit{NULL}\,);$$

The *setlocale*() function returns 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*() is a pointer to a static area within the function and is not guaranteed to remain unchanged [i.e., 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 will set 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. 8859");
```

In this example, all categories of the international environment will be, set to the locale corresponding to the string `"Fr_FR. 8859"`, or to the French language as spoken in France using the ISO 8859-1 code set.

If the string does not correspond to a valid locale, *setlocale*() will return a **NULL** pointer and the international environment is not changed. Otherwise, *setlocale*() will return the name of the locale just set.

`setlocale`(*category*, `"C"`)

The C Standard {2} 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 will set a specific category to an implementation-defined default. For POSIX.1-based systems, this corresponds to the value of the environment variables.

B.8.2 C Language Input/Output Functions

B.8.2.1 Map a Stream Pointer to a File Descriptor

Without some specification of which file descriptors are associated with these streams, it is impossible for an application to set up the streams for another application it starts with *fork*() and *exec*. In particular, it would not be possible to write a portable version of the sh command interpreter (although there may be other constraints that would prevent that portability).

B.8.2.2 Open a Stream on a File Descriptor

The file descriptor may have been obtained from *open*(), *creat*(), *pipe*(), *dup*(), or *fcntl*(); inherited through *fork*() or *exec*; or perhaps obtained by implementation-dependent means, such as the 4.3BSD *socket*() call.

The meanings of the *type* 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. There is no need for `"b"` in the format due to the equivalence of binary and text files in POSIX.1. See B.1.1.1. Although not explicitly required by POSIX.1, a good implementation of append (`"a"`) mode would cause the O_APPEND flag to be set.

B.8.2.3 Interactions of Other *FILE*-Type C Functions

Note that the existence of open streams on a file implies open file descriptors and thus affects the timestamps of the file. The intent is that using *stdio* routines to read a file must eventually update the access time, and using them to write a file must eventually update the modify and change times. However, the exact timing of marking the *st_atime*, *st_ctime*, and *st_mtime* fields cannot be specified, as that would imply a particular buffering strategy.

The purpose of the rules about handles is to allow the writing of a program that uses *stdio* and does some shell-like things; in particular, creating an open file for a child process to use, where both the parent and child wish to use *stdio*, with the consequences of buffering. In most cases, this cannot happen in the C Standard {2} (because there is no way to create a second handle), but the *system*() function can cause this to occur, at least in most historical implementations.

Presently, POSIX.1 deals mostly with output streams; input is implementation defined. It should be possible to make input on seekable devices work for seekable files without affecting buffering strategies significantly. However, the details have not been worked out fully and will be addressed in a future revision of POSIX.1. The requirements on applications are unlikely to change [basically, serving notice to the implementation that the use of a particular handle is (temporarily) completed] and are symmetric to those for output.

There are some implied rules about interprocess synchronization, but no mechanism is given, intentionally. In the simplest case, if the parent meets the requirements on all its files and then performs a *fork*() and a *wait*() before further activity on them [and a *fflush*() on input files after that], the desired synchronization will be achieved. Synchronization could in theory be done with signals, but the only likely case is the one just described. The terms *handle* and *active handle* were required to make the text readable and are not intended for use outside this discussion.

Note that since *exit*() implies *_exit*(), a file descriptor is also closed by *exit*().

Because a handle is either freshly opened, or if not must have handed off control of the open file description as specified, the new handle is always ready to be used (except for seeks) with no initialization. [A freshly opened stream has not yet done any reads, as required by the C Standard {2}, at least implicitly by the rules associated with *setvbuf*().]

In requiring the seek to an appropriate location for the new handle, the application is required to know what it is doing if it is passing streams with seeks involved. If the required seek is not done, the results are undefined (and in fact the program probably will not work on many common implementations).

A naive program used as a utility can be reasonably expected to work properly when the constraints are met by the calling program because it will not hand off file descriptors except with closes.

The *exec* functions are treated specially because the application should always *fflush*() everything before performing one of the *exec* functions. If *stdout* is available on the same open file description after the *exec*, it is a different stream, at least because any unflushed data will be discarded during the *exec* (similarly for *stdin*). Process termination is also special because a process terminating due to a signal or *_exit*() will not have the buffers flushed.

The *fork*() function also must be specially treated because it clones a number of file descriptors simultaneously. Thus, all of them should be prepared for handoff before the *fork*(). In effect, *fork*() creates a pair of handles that are improperly dealt with unless, before the *fork*(), the first part of a handoff occurred. Note that *fflush*(**NULL**) in the C Standard {2} is an appropriate way to do this for output. A subsequent *exec* call [that does not succeed in calling *exit*() in some way] will reduce the number of handles back to the original value (allowing for files that are not close-on-exec), and, thus, preparations for *exec* need not necessarily do the flush. However, because *exit*() closes all streams, if the *exec* fails, the application must be careful to terminate with *_exit*().

POSIX.1 does not specify asynchronous I/O, and when dealing with asynchronous I/O the problem of coordinating access to streams will be more difficult. If asynchronous I/O is provided as an extension, the problems it introduces in this area should be addressed as part of that extension.

It may be that functions such as *system*() and *popen*(), currently being considered for ISO/IEC 9945-2 {B36} , will have to perform some of these operations.

The introduction of underling functions allows generic reference to *errno* values returned by those functions and also to other side effects (as required in the *handles* discussion above). It is not intended to specify implementation, although many implementations may in fact use those functions. The C Standard {2} says very little about *errno* in the context of *stdio*. In the more restricted POSIX.1 environment, providing a reasonable set of *errno* values become possible.

B.8.2.3.1 *fopen*()

There is no additional rationale provided for this subclause.

B.8.2.3.2 *fclose*()

The *fclose*() function is required to synchronize the buffer pointer with the file pointer (unless it already is, which would be the case at EOF). Functionality equivalent to

```
        fseek(stream, ftell(stream), SEEK_SET)
```

does this nicely. The exception for devices incapable of seeking is an obvious requirement, but the implication is that there is no way to reliably read a buffered pipe and hand off handles. This is the situation in historical implementations and is inherent in any “read-ahead” buffering scheme. This limitation is also reflected in the handle hand-off rules.

Note that the last byte read from a stream and the last byte read from an open file description are not necessarily the same; in most cases the open file description's pointer will be past that of the stream because of the stream's read-ahead.

B.8.2.3.3 *freopen*()

There is no additional rationale provided for this subclause.

B.8.2.3.4 *fflush*()

There is no additional rationale provided for this subclause.

B.8.2.3.5 *fgetc*(), *fgets*(), *fread*(), *getc*(), *getchar*(), *gets*(), *scanf*(), *fscanf*()

There is no additional rationale provided for this subclause.

B.8.2.3.6 *fputc*(), *fputs*(), *fwrite*(), *putc*(), *putchar*(), *puts*(), *printf*(), *vprintf*(), *vfprintf*()

There is no additional rationale provided for this subclause.

B.8.2.3.7 *fseek*(), *rewind*()

The *fseek*() function must operate as specified to make the case where seeking is being done work. The key requirement is to avoid an optimization such that an *fseek*() would not result in an *lseek*() if the *fseek*() pointed within the current buffer. This optimization is valuable in general, so it is only required after an *fflush*().

B.8.2.3.8 *perror*()

There is no additional rationale provided for this subclause.

B.8.2.3.9 *tmpfile*()

There is no additional rationale provided for this subclause.

B.8.2.3.10 *ftell*()

In append mode, a *fflush*() will change the seek pointer because of possible writes by other processes on the same file. An *fseek*() reflects the underlying file's file offset, which is not necessarily the end of the file. Implementors should be aware that the operating system itself (not some in-memory approximation) of the file offset should be queried when in append mode.

B.8.2.3.11 Error Reporting

POSIX.1 intentionally does not require that all errors detected by the underlying functions be detected by the functions listed here. There are many reasonable cases where this might not occur; for example, many of the functions with *write*() as an underlying function might not detect a number of error conditions in cases where they simply buffer output for a subsequent flush.

[ENOMEM] was considered for addition as an explicit possible error because most implementations use *malloc*(). This was not done because the scope does not include “out of resource” errors. Nevertheless this is the most likely error to be added to the possible error conditions. Other implementation-defined errors, particularly in the *f*open*() family, are to be expected, and the generic rules about adding (or deleting) possible errors apply, except that it is expected that implementation-defined changes in the error set returned by *open*() would also apply to *fopen*() [unless the condition cannot possibly happen in *fopen*(), which may be possible, but appears unlikely].

B.8.2.3.12 *exit*(), *abort*()

POSIX.1 intends that processing related to the *abort*() function will occur unless “the signal SIGABRT is being caught, and the signal handler does not return,” as defined by the C Standard {2}. This processing includes at least the effect of *fclose*() on all open streams, and the default actions defined for SIGABRT.

The *abort*() function will override blocking or ignoring the SIGABRT signal. Catching the signal is intended to provide the application writer with a portable means to abort processing, free from possible interference from any implementation-provided library functions.

Note that the term “program termination” in the C Standard {2} is equivalent to “process termination” in POSIX.1.

B.8.2.4 Operations on Files — the *remove*() Function

There is no additional rationale provided for this subclause.

B.8.2.5 Temporary File Name — the *tmpnam*() Function

Portable applications that use threads cannot call *tmpnam*() with **NULL** as the parameter if either {_POSIX_THREAD_SAFE_FUNCTIONS} or {_POSIX_THREADS} is defined.

If *s* is not **NULL,** the *tmpnam*() function generates a string that is a valid file name and that is not the same as the name of an existing file.

The *tmpnam*() function generates a different string each time it is called, up to TMP_MAX times. If it is called more than TMP_MAX times, the behavior is implementation defined.

B.8.2.6 Stdio Locking Functions

The *flockfile*() and *funlockfile*() functions provide an orthogonal mutual exclusion lock for each FILE. The *ftrylockfile*() function provides a nonblocking 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 instance, 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 will ensue. 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 will 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.

B.8.2.7 Stdio With Explicit Client Locking

Some I/O functions are typically implemented as macros for performance reasons [for example, *putc*() and *getc*()]. For safety, they need to be synchronized, but it is often too expensive to synchronize on every character. Nevertheless, it was felt that the safety concerns were more important; consequently, the current functions are required to be thread-safe. However, unlocked versions are also provided with names that clearly indicate the unsafe nature of their operation but can be used to exploit their higher performance. These unlocked versions can be safely used only within explicitly locked program regions, using exported locking primitives. In particular, a sequence such as

```
flockfile(fileptr);
putc_unlocked('1', fileptr);
putc_unlocked('\n', fileptr);
```

```
        fprintf(fileptr, "Line 2\n");
        funlockfile (fileptr);
```

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*(), etc., map to the “faster, but unsafe” rather than the “slower, but safe” versions. In either case, one would still want to inspect all uses of *getc*(), *putc*(), etc., 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 interface” invariant. To do otherwise would introduce gratuitous synchronization errors into converted code. Other routines that modify the *stdio* (*FILE* *) structures or buffers will also be safely synchronized.

Note that there is no need for functions of the form *getc_locked*(), *putc_locked*(), etc., 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 C Standard {2} 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 to be executed synchronously.

The *ungetc*() function is infrequently called relative to the other functions/macros so no unlocked variation is needed.

B.8.3 Other C Language Functions

B.8.3.1 Nonlocal Jumps

The C Standard {2} 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 *sig-setjmp*() macro.

There are processors that cannot easily support these calls, but this was not considered a sufficient reason to exclude them.

The distinction between *setjmp*()/*longjmp*() and *sigsetjmp*()/*siglongjmp*() is only significant for programs that use the *sigaction*(), *sigprocmask*(), or *sigsuspend*() functions. Since earlier implementations did not have signal masks, only a single pair was provided.

4.2BSD and 4.3BSD systems provide functions named *_setjmp*() and *_longjmp*() that, together with *setjmp*() and *longjmp*(), provide the same functionality as *sig-setjmp*() 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* {B41} , *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 edition of this standard provided the matching *setjmp*() or *sigsetjmp*() has been performed in the same thread. Nonlocal jumps into contexts saved by other threads would be at best a questionable practice and were not considered worthy of standardization.

B.8.3.2 Set Time Zone

There is no additional rationale provided for this subclause.

B.8.3.3 Find String Token

The *strtok*() function searches for a separator string within a larger string. It returns a pointer to the last substring between separator strings. This function uses static storage to keep track of the current string position between calls. The new function, *strtok_r*(), takes an additional argument, *lasts*, to keep track of the current position in the string.

B.8.3.4 ASCII Time Representation

Note that the C Standard {2} specifies 26 B as the length of the string.

B.8.3.5 Current Time Representation

Note that the C Standard {2} specifies 26 B as the length of the string.

B.8.3.6 Coordinated Universal Time

There is no additional rationale provided for this subclause.

B.8.3.7 Local Time

There is no additional rationale provided for this subclause.

B.8.3.8 Pseudo-Random Sequence Generation Functions

The C Standard {2} *rand*() and *srand*() functions allow per-process pseudo-random streams shared by all threads. Those two functional interfaces 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 multithreaded 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 interface based on whether the seed value is global to the entire process or local to each thread.

This does not address the known deficiencies of the *rand*() function implementations, which have been approached by maintaining more state. In effect, this specifies new thread-safe forms of a deficient interface. Since alternatives to *rand*() are not standardized, they are not modified as part of this standard.

B.8.3.9 Omitted Memory Management

The *brk*() and *sbrk*() functions frequently were proposed for inclusion in POSIX.1, but they were excluded deliberately. See also B.1.1. The rationale for including them is usually addressed to the argument that it is the *sbrk*() primitive that makes it possible to implement some more general heap management system, such as that provided for C by *malloc*(). The need for such functionality is fully understood, but specifying it as a part of a standard would have the effect of limiting the number of architectures that could support POSIX.1. It might also constrain languages whose memory-management model was not served by the *sbrk*() model.

Memory management is not excluded from POSIX.1: POSIX.1 relies on the language to provide it, and in the C binding (as reflected in Section 8) it is provided by *malloc*(). It would be provided by *new*() in Pascal. In a language like FORTRAN, which does not supply memory management to the user, it would be undesirable to force the language binding to attempt to include such a function. It is reasonable to imagine a language that required a more powerful primitive than *sbrk*() to be implemented, and standardizing *sbrk*() would only constrain such future languages.

POSIX.1 is silent about mixed languages. Mixing languages that provide incompatible memory-management mechanisms can yield unpredictable results. Future standards that address mixing of languages should consider this issue.

Architectures that could not support *sbrk*() are also a limiting factor. In particular, architectures that do not present a model of a single linear address space would be severely constrained by *sbrk*(), but are not so constrained by *malloc*() or *new*().

Each language should specify the memory-management primitives best suited to that language. Whether the implementor chooses to use a more primitive mechanism to implement that, or the implementor chooses to directly implement the language function in the kernel, is not a proper concern of the developers of POSIX.1, nor should it be for any portable application. An application that presumes the *sbrk*() model of memory management will not port to all architectures in any case, for the same reasons that *sbrk*() itself does not work on those architectures. No true gain in application portability would be achieved by mandating such an interface. This implies that an implementor of software that wishes to port to multiple platforms and that attempts to implement its own memory management rather than relying on language-supplied functions must be prepared to deal with multiple platform-supplied primitives and, because it is doing its own memory management inherently, cannot be considered, or be made to be, portable in that regard.

B.9 System Databases

At one time, this section was entitled Passwords, but this title was changed as all references to a “password file” were changed to refer to a “user database.”

B.9.1 System Databases

There are no references in POSIX.1 to a *passwd file* or a *group file*, and there is no requirement that the *group* or *passwd* databases be kept in files containing editable text. Many large timesharing systems use *passwd* databases that are hashed for speed. Certain security classifications prohibit certain information in the *passwd* database from being publicly readable.

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

The *getgrent*(), *setgrent*(), *endgrent*(), *getpwent*(), *setpwent*(), and *endpwent*() functions are not included in POSIX.1 because they provide a linear database search capability that is not generally useful [the *getpwuid*(), *getpwnam*(), *getgrgid*(), and *getgrnam*() functions are provided for keyed lookup] and because in certain distributed systems, especially those with different authentication domains, it may not be possible or desirable to provide an application with the ability to browse the system databases indiscriminately.

A change from historical implementations is that the structures used by these functions have fields of the types *gid_t* and *uid_t*, which are required to be defined in the header `<sys/types.h>`. POSIX.1 has not changed the synopses of these functions to require the inclusion of this header, since that would invalidate a large number of existing applications. Implementations must ensure that these types are defined by the inclusion of `<grp.h>` and `<pwd.h>`, respectively, without imposing any namespace pollution or errors from redefinition of types.

POSIX.1 is silent about the content of the strings containing user or group names. These could be digit strings. POSIX.1 is also silent as to whether such digit strings bear any relationship to the corresponding (numeric) user or group ID.

B.9.2 Database Access

B.9.2.1 Group Database Access

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

B.9.2.2 User Database Access

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

B.10 Data Interchange Format

B.10.1 Archive/Interchange File Format

There are three areas of interest associated with file interchange:

1) *Media*. There are other existing standards that define the media used for data interchange.
2) *User Interface*. This rightfully should be in the shell and utilities standard, under development as ISO/IEC 9945-2 {B36} .
3) *Format of the Data*. None of the groups currently developing POSIX standards address topics that match this area. The groups felt that this area is closest to the types of things that should be in the POSIX.1 document, as the level of that document most closely matches the level of data required.

There are two programs in wide use today: `tar` and `cpio`. There are many supporters for each program. Four options were considered for POSIX.1:

1) Make both formats optional. This was considered unacceptable because it does not allow any portable method for data interchange.
2) Require one format.
3) Require one format with the other optional.
4) Require both formats.

Both the Extended `cpio` and the Extended `tar` Formats are required by POSIX.1.

There are a number of concerns about defining extensions that are known to be required by historical implementations. Failure to specify a consistent method to implement these extensions will limit portability of the data and, more importantly, will create confusion if these extensions are later standardized.

Two of these extensions that should be documented are symbolic links, which were defined by 4.2BSD and 4.3BSD systems, and high-performance (or contiguous) files, which exist in a number of implementations and are now being considered for future amendments to POSIX.1.

By defining these extensions, implementors are able to recognize these features and take appropriate implementation-defined actions for these files. For example, a high-performance file could be converted to a regular file if the system did not support high-performance files; symbolic links might be replaced by normal hard links.

The policy of not defining user interfaces to utilities preempted any description of a `tar` or `cpio` command. The behavior of the former command was described in some detail in previous drafts.

The possibilities for transportable media include, but are not limited to

1) 12,7 mm (0,5 in) magnetic tape, 9 track, 63 bpmm (1 600 bpi)
2) 12,7 mm (0,5 in) magnetic tape, 9 track, 246 cpmm (6 250 cpi)
3) QIC-11, 6,30 mm (0,25 in) streamer tape
4) QIC-24, 6,30 mm (0,25 in) streamer tape
5) 130 mm (5,25 in) diskettes, 9 512-byte sectors/track, 3,8 tpmm (96 tpi)
6) 130 mm (5,25 in) diskettes, 9 512-byte sectors/track, 1,9 tpmm (48 tpi)

When selecting media, issues such as character frame size also need to be addressed. The easiest environment for interchange occurs when 8-bit frames are used.

The utilities are not restricted to work only with *transportable* media: existing related utilities are often used to transport data from one place to another in the file hierarchy.

The formats are included to provide implementation-independent ways to move files from one system to another and also to provide ways for a user to save data on a transportable medium to be restored at a later date. Unfortunately, these two goals can contradict each other, as system security problems are easy to find in tape systems if they are not protected. Thus, there are strict requirements about how the mechanism to copy files shall react when operated by both privileged and nonprivileged users. The general concept is that a privileged (historically using the ISUID bit in the file's mode with the owner UID of the file set to super-user) version of the utility (or one operated by a privileged user) can be used as a save/restore scheme, but a nonprivileged version is used to interpret media from a different system without compromising system security.

Regardless of the archive format used, guidelines should be observed when writing tapes to be read on other systems. Assuming the target system conforms to POSIX.1, archives created should only use definitions found in POSIX.1 (e.g., file types, minimum values as found in Section 2) and should only use relative pathnames (i.e., no leading slash).

Both `tar` and `cpio` formats have traditionally been used for both exchange of information and archiving. These formats have a number of features that facilitate archiving, for example, the ability to store information about a file that is a device. POSIX.1 does not assume this kind of data is portable. It is intended that these formats provide for the portable exchange of source information between dissimilar systems. This requires specification of the character set to be used (ISO/IEC 646 {1}) when these formats are used to write source information. The 1990 version of ISO/IEC 646 {1} IRV was selected as the international character set that corresponds most directly to the ASCII set used in many historical implementations. The 1990 version was chosen over the 1983 version because it defines '$' as the currency symbol in the IRV, as opposed to the starburst-like generic currency symbol. Note that ISO/IEC 646 {1} is a safe lowest-common-denominator character set and that interchange of larger character sets is permitted by mutual agreement. Using any other character set (such as ISO 8859-1 {B34} or some multibyte character set) reduces the number of machines to which interchange is guaranteed.

All data written by format-creating utilities and read by format-reading utilities is an ordered stream of bytes. The first byte of the stream should be first on the medium, the second byte second, etc. On systems where the hardware swaps bytes or otherwise rearranges the byte stream on output or input, the implementor of these utilities must compensate for this so that the data on the storage device retains its ordered nature.

POSIX.1 describes two different formats for data archiving and interchange. Strong support for both formats was evident through the balloting process. This is a clear indication of the need for both formats due to existing practice. The balloting process also defined a number of deficiencies of each format. The strong support indicates that these deficiencies are not sufficient to remove either format from POSIX.1, but will need to be addressed in future amendments to POSIX.1. It was not practical to remedy these deficiencies during the balloting process. Considerable thought and review must occur before making any changes to these formats. It was felt that the best solution is to advise implementors and application writers of these deficiencies by documenting them in the rationale and to include both formats in POSIX.1.

The developers of POSIX.1 recognize the desirability for migration toward one common format and have been made aware of some strong inputs to consider both formats in light of existing practice, current technology trends, and the POSIX standards activities such as security and high-performance systems, and to develop one format that is technically superior. This format will be incorporated into a future amendment to POSIX.1 when it is developed.

The deficiencies that have been identified in the existing formats are as follows. The size of a file link is limited to 100 characters in `tar`. A number of fields in the `cpio` header (*c_filesize*, *c_dev*, *c_ino*, *c_mode*, and *c_rdev*) are too short to support values that POSIX.1 allows these fields to contain. Some existing implementations and current trends in development will require the ability to represent even larger values in these fields. The `cpio` format does not provide a mechanism to represent the user and group IDs symbolically, and a range of implementation-defined file types have not been reserved for the user. The `cpio` format specification does not reserve any formats for implementation-defined usage. The extensions that have been made to `cpio` for POSIX.1 are compatible with existing versions of `cpio`. Correction of some of these deficiencies would make existing versions of `cpio` behave unpredictably. When these changes are made the `cpio` magic number will have to be changed.

This clause uses the term *file name*; note that *filename* and *file name* are not synonyms; the latter is a synonym for *pathname*, in that it includes the slashes between filenames.

In earlier drafts, the word “local” was used in the context of “file system” and was taken (incorrectly) to be related to “remotely mounted file system.” This was not intended. The term “(local) file system” refers to the file hierarchy as seen by the utilities, and “local” was removed because of this confusion.

B.10.1.1 Extended `tar` Format

The original model for this facility is the 4.3BSD or Version 7 `tar` program and format, but the format given here is an extension of the traditional `tar` format. The name USTAR was adopted to reflect this.

This description reflects numerous enhancements over previous versions. The goal of these changes was not only to provide the functional enhancements desired, but also to retain compatibility between new and old versions. This compatibility has been retained. Archives written using the old archive format are compatible with the new format. Archives written using this new format may be read by applications designed to use the old format as long as the functional enhancements provided here are not used. This means the user is limited to archiving only regular type files and nonsymbolic links to such files.

Implementors should be aware that the previous file format did not include a mechanism to archive directory type files. For this reason, the convention of using a file name ending with slash was adopted to specify a directory on the archive.

The total size of the name and prefix fields have been set to meet the minimum requirements for {PATH_MAX}. If a pathname will fit within the name field, it is recommended that the pathname be stored there without the use of the prefix field. Although the name field is known to be too small to contain {PATH_MAX} characters, the value was not changed in this version of the archive file format to retain backward compatibility, and instead the *prefix* was introduced. Also, because of the earlier version of the format, there is no way to remove the restriction on the *linkname* field being limited in size to just that of the *name* field.

The *size* field is required to be meaningful in all implementation extensions, although it could be zero. This is required so that the data blocks can always be properly counted.

It is suggested that if device special files need to be represented that cannot be represented in the standard format that one of the extension types (`'A'-'Z'`) be used, and that the additional information for the special file be represented as data and be reflected in the size field.

Attempting to restore a special file type, where it is converted to ordinary data and conflicts with an existing file name, need not be specially detected by the utility. If run as an ordinary user, a format-reading utility should not be able to overwrite the entries in, for example, `/dev` in any case (whether the file is converted to another type or not). If run as a privileged user, it should be able to do so, and it would be considered a bug if it did not. The same is true of ordinary data files and similarly named special files; it is impossible to anticipate the user's needs (who could really intend to overwrite the file), so the behavior should be predictable (and thus regular) and rely on the protection system as required.

The values `'2'` and `'7'` in the typeflag field are intended to define how symbolic links and contiguous files can be stored in a `tar` archive. POSIX.1 does not require the symbolic link or contiguous file extensions, but does define a standard way of archiving these files so that all conforming systems can interpret these file types in a meaningful and consistent manner. On a system that does not support extended file types, the format-interpreting utility should do the best it can with the file and go on to the next.

B.10.1.2 Extended `cpio` Format

The model for this format is the existing System V `cpio –c` data interchange format. This model documents the portable version of `cpio` format and not the binary version. It has the flexibility to transfer data of any type described within the POSIX.1 standard, yet is extensible to transfer data types specific to extensions beyond POSIX.1 (e.g., symbolic links or contiguous files). Because it describes existing practice, there is no question of maintaining upward compatibility.

This subclause does not standardize behavior for the utility when the file type is not understood or supported. It is useful for the utility to report to the user whatever action is taken in this case, though POSIX.1 neither requires nor recommends this.

B.10.1.2.1 `cpio` Header

There has been some concern that the size of the *c_ino* field of the header is too small to handle those systems that have very large i-node numbers. However, the *c_ino* field in the header is used strictly as a hard link resolution mechanism for archives. It is not necessarily the same value as the i-node number of the file in the location from which that file is extracted.

B.10.1.2.2 `cpio` File Name

For most historical implementations of the `cpio` utility, {PATH_MAX} bytes can be used to describe the pathname without the addition of any other header fields (the null byte would be included in this count). {PATH_MAX} is the minimum value for pathname size, documented as 256 bytes in Section 2 However, an implementation may use *c_namesize* to determine the exact length of the pathname. With the current description of the `cpio` header, this pathname size can be as large as a number that is described in six octal digits.

B.10.1.2.3 `cpio` File Data

There is no additional rationale provided for this subclause.

B.10.1.2.4 `cpio` Special Entries

These are provided to maintain backward compatibility.

B.10.1.2.5 `cpio` Values

Three values are documented under the *c_mode* field values to provide for extensibility for known file types:

0110 000	Reserved for contiguous files. The implementation may treat the rest of the information for this archive like a regular file. If this file type is undefined, the implementation may create the file as a regular file.

0120 000	Reserved for files with symbolic links. The implementation may store the link name within the data portion of the file. If this type is undefined, the implementation may not know how to link this file or be able to understand the data section. The implementation may decide to ignore this file type and output a warning message.
0140 000	Reserved for sockets. If this type is undefined on the target system, the implementation may decide to ignore this file type and output a warning message.

This provides for extensibility of the `cpio` format while allowing for the ability to read old archives. Files of an unknown type may be read as "regular files" on some implementations. On a system that does not support extended file types, the format-interpreting utility should do the best it can with the file and go on to the next.

B.10.1.3 Multiple Volumes

Multivolume archives have been introduced in a manner that has become a *de facto* standard in many implementations. Though it is not required by POSIX.1, classical implementations of the format-reading and -creating utility, upon reading logical end-of-file, check to see if an error channel is open to a controlling terminal. The utility then produces a message requesting a new medium to be made available. The utility waits for a new medium to be made available by attempting to read a message to restart from the controlling terminal. In all cases, the communication with the controlling terminal is in an implementation-defined manner.

This subclause (10.1.3) is intended to handle the issue of multiple volume archives. Since the end-of-medium and transition between media are not properly part of POSIX.1, the transition is described in terms of files; the word "file" is used in a very broad, but correct, sense—a tape drive is a file. The intent is that files will be read serially until the end-of-archive indication is encountered and that file or media change will be handled by the utilities in an implementation-defined manner.

Note that there was an issue with the representation of this on magnetic tape, and POSIX.1 is intended to be interpreted such that each byte of the format is represented on the media exactly once. In some current implementations, it is not deterministic whether encountering the end-of-medium reflector foil on magnetic tape during a write will yield an error during a subsequent *read*() of that record, or if that record is actually recorded on the tape. It is also possible that *read*() will encounter the end-of-medium when end-of-medium was not encountered when the data was written. This has to do with conditions where the end of [magnetic] record is in such a position that the reflector foil is on the verge of being detected by the sensor and is detected during one operation and not on a later one, or vice versa.

An implementation of the format-creating utility must assure when it writes a record that the data appears on the tape exactly once. This implies that the program and the tape driver work in concert. An implementation of the format-creating utility must assure that an error in a boundary condition described above will not cause loss of data.

The general consensus was that the following would be considered as correct operation of a tape driver when end-of-medium is detected:

1) During writing, either
   a) The record where the reflector spot was detected is backspaced over by the driver so that the trailing tape mark that will be written on *close*() will overwrite. Writing the tape mark should not yield an end-of-medium condition, or
   b) The condition is reported as an error on the *write*() following the one where the end-of-medium is detected (the one where the end-of-medium is actually detected completing successfully). No data will be actually transferred on the *write*() reporting the error. The subsequent *close*() would write a tape mark following the last record actually written. Writing the tape mark, and writing any subsequent records, should not yield any end-of-medium conditions.

   [The latter behavior permits the implementation of ANSI standard labels because several records (the trailer records) can be written after the end-of-medium indications. It also permits dealing with, for example, COBOL "ON" statements.

2) During reading, the end-of-medium indicator is simply ignored, presuming that a tape mark (end-of-file) will be recorded on the magnetic medium and that the reflector foil was advisory only to the *write*().

Systems where these conditions are not met by the tape driver should assure that the format-creating and -reading utilities assure proper representation and interpretations of the files on the media in a way consistent with the above recommendations.

The typical failures on systems that do not meet the above conditions are either

1) To leave the record written when the end-of-medium is encountered on the tape, but to report that it was not written. The format-creating utility would then rewrite it, and then the format-reading utility could see the record twice if the end-of-medium is not sensed during the read operations, or
2) The *write*() occurs uneventfully, but the *read*() senses the error and does not actually see the data, causing a record to be omitted.

Nothing in POSIX.1 requires that end-of-medium be determined by anything on the medium itself (for example, a predetermined maximum size would be an acceptable solution for the format-creating utility). The format-reading utility must be able to *read*() tapes written by machines that do use the whole medium, however.

On media where end-of-medium and end-of-file are reliably coincident, such as disks, end-of-medium and end-of-file can be treated as synonyms.

Note that partial physical records [corresponding to a single *write*()] can be written on some media, but that only full physical records will actually be written to magnetic tape, given the manner in which the tape operates.

B.11 Synchronization

B.11.1 Semaphore Characteristics

There is no specific rationale for this subclause.

B.11.2 Semaphore Functions

Semaphores are a high-performance process synchronization mechanism. Semaphores are named by null-terminated strings of characters.

A semaphore is created using the *sem_init*() function of the *sem_open*() function with the O_CREAT flag set in oflag.

To use a semaphore, a process has to first initialize the semaphore or inherit an open descriptor for the semaphore via *fork*()

A semaphore preserves its state when the last reference is closed. For example, if a semaphore has a value of 13 when the last reference is closed, it will have a value of 13 when it is next opened.

When a semaphore is created, an initial state for the semaphore has to be provided. This value is a nonnegative integer. Negative values are not possible since they indicate the presence of blocked processes. The persistence of any of these objects across a system crash or a system reboot is undefined. Conforming applications shall not depend on any sort of persistence across a system reboot or a system crash.

Models and Requirements

A realtime system requires synchronization and communication between the processes comprising the overall application. An efficient and reliable synchronization mechanism has to be provided in a realtime system that will allow more than one schedulable process mutually exclusive access to the same resource. This synchronization mechanism has to allow for the optimal implementation of synchronization or systems implementors will define other, more cost-effective methods.

At issue are the methods whereby multiple processes (tasks) can be designed and implemented to work together in order to perform a single function. This requires interprocess communication and synchronization. A semaphore mechanism is the lowest level of synchronization that can be provided by an operating system.

A semaphore is defined as an object that has an integral value and a set of blocked processes associated with it. If the value is positive or zero, then the set of blocked processes is empty; otherwise, the size of the set is equal to the absolute value of the semaphore value. The value of the semaphore can be incremented or decremented by any process with access to the semaphore and must be done as an indivisible operation. When a semaphore value is less than or equal to zero, any process that attempts to lock it again will block or be informed that it is not possible to perform the operation.

A semaphore may be used to guard access to any resource accessible by more than one schedulable task in the system. It is a global entity and not associated with any particular process. As such, a method of obtaining access to the semaphore has to be provided by the operating system. A process that wants access to a critical resource (section) has to wait on the semaphore that guards that resource. When the semaphore is locked on behalf of a process, it knows that it can utilize the resource without interference by any other cooperating process in the system. When the process finishes its operation on the resource, leaving it in a well-defined state, it posts the semaphore, indicating that some other process may now obtain the resource associated with that semaphore.

In this section, mutexes and condition variables are specified as the synchronization mechanisms between threads.

These primitives are typically used for synchronizing threads that share memory in a single process. However, this section provides an option allowing the use of these synchronization interfaces and objects between processes that share memory, regardless of the method for sharing memory.

Much experience with semaphores shows that there are two distinct uses of synchronization: locking, which is typically of short duration; and waiting, which is typically of long or unbounded duration. These distinct usages map directly onto mutexes and condition variables, respectively.

Semaphores are provided in this standard primarily to provide a means of synchronization for processes; these processes may or may not share memory. Mutexes and condition variables are specified as synchronization mechanisms between threads; these threads always share (some) memory. Both are synchronization paradigms that have been in widespread use for a number of years. Each set of primitives is particularly well matched to certain problems.

With respect to binary semaphores, experience has shown that condition variables and mutexes are easier to use for many synchronization problems than binary semaphores. The primary reason for this is the explicit appearance of a Boolean predicate that specifies when the condition wait is satisfied. This Boolean predicate terminates a loop, including the call to *pthread_cond_wait*(). As a result, extra wakeups are benign since the predicate governs whether the thread will actually proceed past the condition wait. With stateful primitives, such as binary semaphores, the wakeup in itself typically means that the wait is satisfied. The burden of ensuring correctness for such waits is thus placed on *all* signalers of the semaphore rather than on an *explicitly coded* Boolean predicate located at the condition wait. Experience has shown that the latter creates a major improvement in safety and ease of use.

Counting semaphores are well matched to dealing with producer/consumer problems, including those that might exist between threads of different processes, or between a signal handler and a thread. In the former case, there may be little or no memory shared by the processes; in the latter case, one is not communicating between co-equal threads, but between a thread and an interruptlike entity. It is for these reasons that this standard allows semaphores to be used by threads.

Mutexes and condition variables have been effectively used with and without priority inheritance, priority ceiling, and other attributes to synchronize threads that share memory. The efficiency of their implementation is comparable to or better than that of other synchronization primitives that are sometimes harder to use (for example, binary semaphores). Furthermore, there is at least one known implementation of Ada tasking that uses these primitives. Mutexes and condition variables together constitute an appropriate, sufficient, and complete set of interthread synchronization primitives.

Efficient multithreaded applications require high-performance synchronization primitives. Considerations of efficiency and generality require a small set of primitives upon which more sophisticated synchronization functions can be built.

Standardization Issues

It is possible to implement very high-performance semaphores using test-and-set instructions on shared memory locations. The library routines that implement such a high-performance interface has to properly ensure that a *sem_wait*() or *sem_trywait*() operation that cannot be performed will issue a blocking semaphore system call or properly report the condition to the application. The same interface to the application program would be provided by a high-performance implementation.

B.11.2.1 Initialize an Unnamed Semaphore

The revised text restricts the use of *sem_init*() to a single initialization of any particular semaphore.

Although this standard fails to specify a successful return value, it is likely that a later amendment may require the implementation to return a value of zero if the call to *sem_init*() is successful.

B.11.2.2 Destroy an Unnamed Semaphore

There is no specific rationale for this subclause.

B.11.2.3 Initialize/Open a Named Semaphore

The previous version of this standard 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.

B.11.2.4 Close a Named Semaphore

There is no specific rationale for this subclause.

B.11.2.5 Remove a Named Semaphore

There is no specific rationale for this subclause.

B.11.2.6 Lock a Semaphore

There is no specific rationale for this subclause.

B.11.2.7 Unlock a Semaphore

There is no specific rationale for this subclause.

B.11.2.8 Get the Value of a Semaphore

There is no specific rationale for this subclause.

B.11.3 Mutexes

B.11.3.1 Mutex Initialization Attributes

See B.16.2.1 for a general explanation of attributes. Attribute objects allow implementations to experiment with useful extensions and permit extension of this standard without changing the existing interfaces. They thus provide for future extensibility of this standard 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, etc.) 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 the standard. 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 nondefault 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 nondefault attributes. Any later attempt to lock such a mutex will cause 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 nondefault 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 standard 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 (i.e., 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 int 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(&semap->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 semphore 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);
}
```

B.11.3.2 Initializing and Destroying a Mutex

Alternate Implementations Possible

The standard 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, the standard 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 overspecification of the type of the mutex or condition variable and motivates the opacity 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, the standard ensures that an application fully conforming to the standard will be portable across the full range of implementations—while not forcing all implementations to add overhead to check for numerous things that a correct program will never do.

Why No Limits 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 will 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 like

```
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

```
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 will also be faster on a large class of systems—those where the (entire) synchronization object can be stored in application memory.

Yet the locking performance question will be raised for machines that require mutexes to be allocated out of special memory. Such machines will 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*() will also have to test whether or not the pointer to the actual lock has been allocated. If it has not, *pthread_mutex_lock*() will have 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 if 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 will have 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 will incur no extra overhead since they will 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 (e.g. lines A and B) immediately after the object is unlocked (line C).

B.11.3.3 Locking and Unlocking a Mutex

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 interfaces and the particular default settings of the mutex attributes have been motivated by the desire to not preclude fast, inlined implementations of mutex locking and unlocking.

For instance, deadlocking on a double-lock is explicitly allowed behavior in order to avoid requiring more overhead in the basic mechanism than is absolutely necessary. (More “friendly” mutexes that detect deadlock or that allow multiple locking by the same thread are easily constructed by the user via the other mechanisms provided. For example, *pthread_self*() can be used to record mutex ownership.) Implementations might also choose to provide such extended features as options via special mutex attributes.

Since most attributes only need to be checked when a thread is going to be blocked, the use of attributes does not slow the (common) mutex-locking case.

Likewise, while being able to extract the thread ID of the owner of a mutex might be desirable, it would require storing the current thread ID when each mutex is locked, and this could incur unacceptable levels of overhead. Similar arguments apply to a “mutex_tryunlock” operation.

B.11.4 Condition Variables

B.11.4.1 Condition Variable Initialization Attributes

See B.16.2.1 and B.11.3.1.

A `process-shared` attribute has been defined for condition variables for the same reason it has been defined for mutexes.

B.11.4.2 Initializing and Destroying Condition Variables

See B.11.3.2; a similar rationale applies to condition variables.

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(1, 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.

B.11.4.3 Broadcasting and Signaling a Condition

Multiple Awakenings by Condition Signal

On a multiprocessor, 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);              /*  9 */
        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);           /*  2 */
        cond->value++;                             /*  3 */
        if (cond->waiter) {                        /*  4 */
                sleeper = cond->waiter;            /*  5 */
                cond->waiter = sleeper->next_cond;  /*  6 */
                able_to_run(sleeper);              /*  7 */
        }
        pthread_mutex_unlock(cond->mutex);         /*  8 */
```

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 will block.

While this problem could be resolved, the loss of efficiency for a fringe condition that will occur 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 will be forced to code a predicate-testing-loop around the condition wait. This will also make 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, the standard explicitly documents that spurious wakeups may occur.

Condition Broadcast

The *pthread_cond_broadcast*() function is used whenever the shared-variable state has been changed in a way that more than one thread can proceed with its task. Consider a single-producer/multiple-consumer problem, where the producer can insert multiple items on a list that is accessed one item at a time by the consumers. By calling the *pthread_cond_broadcast*() function, the producer would notify all consumers that might be waiting, and thereby the application would receive more throughput on a multiprocessor. In addition, *pthread_cond_broadcast*() makes it easier to implement a readers/writer lock. The *pthread_cond_broadcast*() function is needed in order to wake up all waiting readers when a writer releases its lock. Finally, the two-phase commit algorithm can use this broadcast function to notify all clients of an impending transaction commit.

Releasing Threads From a Signal Handler

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 predicate, the signal delivery, and the call to *pthread_cond_wait*() that could not be efficiently eliminated.

Mutexes and condition variables are thus not suitable for releasing a waiting thread by signalling from code running in a signal handler.

To provide a convenient way for a thread to await a signal, this standard provides the function *sigwait*(). For most cases where a thread has to wait for a signal, the function *sigwait*() should be quite convenient, efficient, and adequate.

However, requests were made for a 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*().

B.11.4.4 Waiting on a Condition

Condition Wait Semantics

It is important to note that when *pthread_cond_wait*() and *pthread_cond_timedwait*() return without error, the associated predicate may still be false. Similarly, when *pthread_cond_timedwait*() returns with the timeout error, the associated predicate may be true due to an unavoidable race between the expiration of the timeout and the predicate state change.

Some implementations, particularly on a multiprocessor, may sometimes cause multiple threads to wake up when the condition variable is signaled simultaneously on different processors.

In general, whenever a condition wait returns, the thread has to re-evaluate the predicate associated with the condition wait to determine whether it can safely proceed, should wait again, or should declare a timeout. A return from the wait does not imply that the associated predicate is either true or false.

It is thus recommended that a condition wait be enclosed in the equivalent of a “while loop” that checks the predicate.

Timed Wait Semantics

An absolute time measure was chosen for specifying the timeout parameter for two reasons. First, a relative time measure can be easily implemented on top of an interface that specifies absolute time, but there is a race condition associated with specifying an absolute timeout on top of an interface that specifies relative timeouts. For example, assume that *clock_gettime*() returns the current time and *pthread_cond_relative_timed_wait*() uses relative timeouts:

```
clock_gettime (CLOCK_REALTIME, &now)
reltime = sleep_til_this_absolute_time - now;
cond_relative_timed_wait(c, m, &reltime);
```

If the thread is preempted between the first statement and the last statement, the thread will block for too long. Blocking, however, is irrelevant if an absolute timeout is used. An absolute timeout also need not be recomputed if it is used multiple times in a loop, such as that enclosing a condition wait.

For cases when the system clock is advanced discontinuously by an operator, it is expected that implementations will process any timed wait expiring at an intervening time as if that time had actually occurred.

Cancellation and Condition Wait

A condition wait, whether timed or not, is a cancellation point. That is, the functions *pthread_cond_wait*() or *pthread_cond_timedwait*() are points where a pending (or concurrent) cancellation request is noticed. The reason for this is that an indefinite wait is possible at these points—whatever event is being waited for, even if the program is totally correct, might never occur; for example, some input data being awaited might never be sent. By making condition wait a cancellation point, the thread can be canceled and perform its cancellation cleanup handler even though it may be stuck in some indefinite wait.

A side effect of acting on a cancellation request while a thread is blocked on a condition variable is to reacquire the mutex before calling any of the cancellation cleanup handlers. This is done in order to ensure that the cancellation cleanup handler is executed in the same state as the critical code that lies both before and after the call to the condition wait function. This rule is also required when interfacing to pthreads from languages, such as Ada or C++, which may choose to map cancellation onto a language exception—this rule ensures that each exception handler guarding a critical section can always safely depend upon the fact that the associated mutex has already been locked regardless of exactly where within the critical section the exception was raised. Without this rule, there would not be a uniform rule that exception handlers could follow regarding the lock and so coding would become very cumbersome.

Therefore, since *some* statement has to be made regarding the state of the lock when a cancellation is delivered during a wait, a definition has been chosen that makes application coding most convenient and error free.

When acting on a cancellation request while a thread is blocked on a condition variable, the implementation is required to ensure that the thread does not consume any condition signals directed at that condition variable if there are any other threads waiting on that condition variable. This rule is specified in order to avoid deadlock conditions that could occur if these two independent requests (one acting on a thread and the other acting on the condition variable) were not processed independently.

Performance of Mutexes and Condition Variables

Mutexes are expected to be locked only for a few instructions. This practice is almost automatically enforced by the desire of programmers to avoid long serial regions of execution (which would reduce total effective parallelism).

When using mutexes and condition variables, one tries to ensure that the usual case is to lock the mutex, access shared data, and unlock the mutex. Waiting on a condition variable should be a relatively rare situation. For example, when implementing a readers/writers lock, code that acquires a read-lock typically needs only to increment the count of readers (under mutual exclusion) and return. The calling thread would actually wait on the condition variable only when there is already an active writer. So the efficiency of a synchronization operation is bounded by the cost of mutex lock/unlock and not by condition wait. Note that in the usual case there is no context switch.

This is not to say that the efficiency of condition waiting is unimportant. Since there needs to be at least one context switch per Ada rendezvous, the efficiency of waiting on a condition variable is important. The cost of waiting on a condition variable should be little more than the minimal cost for a context switch plus the time to unlock and lock the mutex.

Features of Mutexes and Condition Variables

It had been suggested that the mutex acquisition and release be decoupled from condition wait. This was rejected because it is the combined nature of the operation that, in fact, facilitates realtime implementations. Those implementations can atomically move a high-priority thread between the condition variable and the mutex in a manner that is transparent to the caller. This can prevent extra context switches and provide more deterministic acquisition of a mutex when the waiting thread is signaled. Thus, fairness and priority issues can be dealt with directly by the scheduling discipline. Furthermore, the current condition wait operation matches existing practice.

Scheduling Behavior of Mutexes and Condition Variables

Synchronization primitives that attempt to interfere with scheduling policy by specifying an ordering rule are considered undesirable. Threads waiting on mutexes and condition variables are selected to proceed in an order dependent upon the scheduling policy rather than in some fixed order (for example, FIFO or priority). Thus, the scheduling policy determines which thread(s) will be awakened and allowed to proceed. See Section 13 for the effect of scheduling policy on synchronization operations.

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.

B.11.4.5 Omitted and Rejected Functions

Adding global names for mutexes and condition variables (such as are provided for semaphores) was considered. This proposed change was rejected because the resulting semantics became overly complicated and error prone. In addition, the change was considered to be inconsistent with the existing practice, where mutexes and condition variables are used exclusively to synchronize threads that access shared memory.

B.12 Memory Management

All memory management and shared memory definitions are located in the header `<sys/mman.h>`. Both the SVR4 and BSD systems use the header `<sys/roman. h>` for memory management definitions.

B.12.1 Memory Locking Functions

This portion of the rationale presents models, requirements, and standardization issues relevant to process memory locking.

Models

POSIX.1b conforming realtime systems are expected (and desired) to be supported on systems with demand-paged virtual memory management, nonpaged swapping memory management, and physical memory systems with no Memory Management hardware. The general case, however, is the demand-paged, virtual memory system with each

POSIX process running in a virtual address space. Note that this includes architectures where each process resides in its own virtual address space and architectures where the address space of each process is only a portion of a larger global virtual address space.

The concept of memory locking is introduced to eliminate the indeterminacy introduced by paging and swapping, and to support an upper bound on the time required to access the memory mapped into the address space of a process. Ideally, this upper bound will be the same as the time required for the processor to access “main memory,” including any address translation and cache miss overheads. But some implementations—primarily on mainframes—will not actually force locked pages to be loaded and held resident in main memory. Rather, they will handle locked pages so that accesses to these pages will meet the performance metrics for locked process memory in the implementation. Also, although it is not, for example, the intention that this interface, as specified, be used to lock process memory into “cache,” it is conceivable that an implementation could support a large static RAM memory and define this as “main memory” and use a large[r] dynamic RAM as “backing store.” These interfaces could then be interpreted as supporting the locking of process memory into the static RAM. Support for multiple levels of backing store would require extensions to these interfaces.

Implementations may also use memory locking to guarantee a fixed translation between virtual and physical addresses where such is beneficial to improving determinancy for direct-to/from-process Input/Output. The standard does not guarantee to the application that the virtual-to-physical address translations, if such exist, are fixed, because such behavior would not be implementable on all architectures on which implementations of this standard are expected. But the standard does mandate that an implementation define, for the benefit of potential users, whether or not locking guarantees fixed translations.

Memory locking is defined with respect to the address space of a process. Only the pages mapped into the address space of a process may be locked by the process, and when the pages are no longer mapped into the address space—for whatever reason—the locks established with respect to that address space are removed. Shared memory areas warrant special mention, as they may be mapped into more than one address space or mapped more than once into the address space of a process; locks may be established on pages within these areas with respect to several of these mappings. In such a case, the lock state of the underlying physical pages is the logical OR of the lock state with respect to each of the mappings. Only when all such locks have been removed are the shared pages considered unlocked.

In recognition of the page granularity of Memory Management Units (MMU), and in order to support locking of ranges of address space, memory locking is defined in terms of “page” granularity. That is, for the interfaces that support an address and size specification for the region to be locked, the address must be on a page boundary, and all pages mapped by the specified range are locked, if valid. This means that the length is implicitly rounded up to a multiple of the page size. The page size is implementation defined and is available to applications as a compile time symbolic constant or at run-time via *sysconf*().

A “real memory” POSIX.1b implementation that has no MMU could elect not to support these interfaces, returning [ENOSYS]. But an application could easily interpret this as meaning that the implementation would unconditionally page or swap the application when such is not the case. It is the intention of the standard that such a system could define these interfaces as “NO-OPs,” returning success without actually performing any function except for mandated argument checking.

Requirements

For realtime applications, memory locking is generally considered to be required as part of application initialization. This locking is performed after an application has been loaded (that is, *exec*'ed) and the program remains locked for its entire lifetime. But to support applications that undergo major mode changes where, in one mode, locking is required, but in another it is not, the specified interfaces allow repeated locking and unlocking of memory within the lifetime of a process.

When a realtime application locks its address space, it should not be necessary for the application to then “touch” all of the pages in the address space to guarantee that they are resident or else suffer potential paging delays the first time the page is referenced. Thus, the standard requires that the pages locked by the specified interfaces be resident when the locking functions return successfully.

Many architectures support system-managed stacks that grow automatically when the current extent of the stack is exceeded. A realtime application has a requirement to be able to "preallocate" sufficient stack space and lock it down so that it will not suffer page faults to grow the stack during critical realtime operation. There was no consensus on a portable way to specify how much stack space is needed, so the standard supports no specific interface for preallocating stack space. But an application can portably lock down a specific amount of stack space by specifying MCL_FUTURE in a call to *memlockall*() and then calling a dummy function that declares an automatic array of the desired size.

Memory locking for realtime applications is also generally considered to be an "all or nothing" proposition. That is, the entire process, or none, is locked down. But, for applications that have well-defined sections that need to be locked and others that do not, the standard supports an optional set of interfaces to lock or unlock a range of process addresses. Reasons for locking down a specific range include

- An asynchronous event handler function that must respond to external events in a deterministic manner such that page faults cannot be tolerated
- An input/output "buffer" area that is the target for direct-to-process I/O, and the overhead of implicit locking and unlocking for each I/O call cannot be tolerated

Finally, locking is generally viewed as an "application-wide" function. That is, the application is globally aware of which regions are locked and which are not over time. This is in contrast to a function that is used temporarily within a "third party" library routine whose function is unknown to the application, and therefore must have no "side effects." The specified interfaces, therefore, do not support "lock stacking" or "lock nesting" within a process. But, for pages that are shared between processes or mapped more than once into a process address space, "lock stacking" is essentially mandated by the requirement that unlocking of pages that are mapped by more that one process or more than once by the same process does not affect locks established on the other mappings.

There was some support for "lock stacking" so that locking could be transparently used in library functions or opaque modules. But the consensus was not to burden all implementations with lock stacking (and reference counting), and an implementation option was proposed. There were strong objections to the option because applications would have to support both options in order to remain portable. The consensus was to eliminate lock stacking altogether, primarily through overwhelming support for the SVR4 "m[un]lock[all]" interface on which the standard is now based.

Locks are not inherited across *fork*()s because some systems implement *fork*() by creating new address spaces for the child. In such an implementation, requiring locks to be inherited would lead to new situations in which a fork would fail due to the inability of the system to lock sufficient memory to lock both the parent and the child. The consensus was that there was no benefit to such inheritance. Note that this does not mean that locks are removed when, for instance, a thread is created in the same address space.

Similarly, locks are not inherited across *execs* because some systems implement *exec* by unmapping all of the pages in the address space (which, by definition, removes the locks on these pages), and maps in pages of the *exec*ed image. In such an implementation, requiring locks to be inherited would lead to new situations in which *exec* would fail. Reporting this failure would be very cumbersome to detect in time to report to the calling process, and no appropriate mechanism exists for informing the *exec*ed process of its status.

It was determined that, if the newly loaded application required locking, it was the responsibility of that application to establish the locks. This is also in keeping with the general view that it is the responsibility of the application to be aware of all locks that are established.

There was one request to allow (not mandate) locks to be inherited across *fork*(), and a request for a flag, MCL_INHERIT, that would specify inheritance of memory locks across *execs*. Given the difficulties raised by this and the general lack of support for the feature in the standard, it was not added. The does not preclude an implementation from providing this feature for administrative purposes, such as a “run” command that will lock down and execute specified program. Additionally, the rationale for the objection equated *fork*() with creating a thread in the address space. The standard does not mandate releasing locks when creating additional threads in an existing process.

Standardization Issues

One goal of the standard is to define a set of primitives that provide the necessary functionality for realtime applications, with consideration for the needs of other application domains where such were identified, which is based to the extent possible on existing industry practice.

The ISO/IEC 9945-1 :1990 definition does not include any memory locking facility. The Memory Locking option is required by many realtime applications to tune performance. Such a facility is accomplished by placing constraints on the virtual memory system to limit paging of time of the process or of critical sections of the process. This facility should not be used by most nonrealtime applications.

Optional features provided in this standard allow applications to lock selected address ranges with the caveat that the process is responsible for being aware of the page granularity of locking and the unnested nature of the locks.

B.12.1.1 Lock/Unlock the Address Space of a Process

There is no specific rationale for this subclause.

B.12.1.2 Lock/Unlock a Range of Process Address Space

There is no specific rationale for this subclause.

B.12.2 Mapped Files Functions

The Memory Mapped Files option provides a mechanism that allows a process to access files by directly incorporating file data into its address space. Once a file is “mapped” into a process address space, the data can be manipulated by instructions as memory. The use of mapped files can significantly reduce I/O data movement since file data does not have to be copied into process data buffers as in *read*() and *write*(). If more than one process maps a file, its contents are shared among them. This provides a low overhead mechanism by which processes can synchronize and communicate.

Historical Perspective

Realtime applications have historically been implemented using a collection of cooperating processes or tasks. In early systems, these processes ran on bare hardware (that is, without an operating system) with no memory relocation or protection. The application paradigms that arose from this environment involve the sharing of data between the processes.

When realtime systems were implemented on top of vendor-supplied operating systems, the paradigm or performance benefits of direct access to data by multiple processes was still deemed necessary. As a result, operating systems that claim to support realtime applications must support the shared memory paradigm.

Additionally, a number of realtime systems provide the ability to map specific sections of the physical address space into the address space of a process. This ability is required if an application is to obtain direct access to memory locations that have specific properties (for example, refresh buffers or display devices, dual ported memory locations, DMA target locations). The use of this ability is common enough to warrant some degree of standardization of its interface. This ability overlaps the general paradigm of shared memory in that, in both instances, common global objects are made addressable by individual processes or tasks.

Finally, a number of systems also provide the ability to map process addresses to files. This provides both a general means of sharing persistent objects, and using files in a manner that optimizes memory and swapping space usage.

Simple shared memory is clearly a special case of the more general file mapping capability. In addition, there is relatively wide spread agreement and implementation of the file mapping interface. In these systems, many different types of objects can be mapped (e.g., files, memory, devices, etc.) using the same mapping interfaces. This approach both minimizes interface proliferation and maximizes the generality of programs using the mapping interfaces.

Memory Mapped Files Usage

A memory object can be concurrently mapped into the address space of one or more processes. The *mmap*() and *munmap*() functions allow a process to manipulate its address space by mapping portions of memory objects into it and removing them from it. When multiple processes map the same memory object, they can share access to the underlying data. Implementations may restrict the size and alignment of mappings to be on page-size boundaries. The page size, in bytes, is the value of the system-configurable variable {PAGESIZE}, typically accessed by calling *sysconf*() with a *name* argument of {_SC_PAGESIZE}. If an implementation has no restrictions on size or alignment, it may specify a 1B page size.

To map memory, a process first opens a memory object. The *ftruncate*() function can be used to contract or extend the size of the memory object even when the object is currently mapped. If the memory object is extended, the contents of the extended areas are zeros.

After opening a memory object, the application maps the object into its address space using the *mmap*() function call. Once a mapping has been established, it remains mapped until unmapped with *munmap*(), even if the memory object is closed. The *mprotect*() function can be used to change the memory protections initially established by *mmap*().

A *close*() of the file descriptor, while invalidating the file descriptor itself, does not unmap any mappings established for the memory object. The address space, including all mapped regions, is inherited on *fork*(). The entire address space is unmapped on process termination or by successful calls to any of the exec functions.

The *msync*() function is used to force mapped file data to permanent storage.

Effects on Other Functions

When the Memory Mapped Files option is supported, the operation of the *open*(), *creat*(), and *unlink*() functions are a natural result of using the file system name space to map the global names for memory objects.

The *ftruncate*() function can be use to set the length of a sharable memory object.

The meaning of *stat*() fields other than the size and protection information is undefined on implementations where memory objects are not implemented using regular files. When regular files are used, the times reflect when the implementation updated the file image of the data, not when a process updated the data in memory.

The operations of *fdopen*(), *write*(), *read*(), and *lseek*() were made unspecified for objects opened with *shm_open*() so that implementations that did not implement memory objects as regular files would not have to support the operation of these functions on shared memory objects.

The behavior of memory objects with respect to *close*(), *dup*(), *dup2*(), *open*(), *close*(), *fork*(), *_exit*(), and *exec* is the same as the behavior of the existing practice of the *mmap*() function.

A memory object can still be referenced after a close. That is, any mappings made to the file are still in effect, and read and writes that are made to those mappings are still valid and are shared with other processes that have the same mapping. Likewise, the memory object can still be used if any references remain after its name(s) have been deleted. Any references that remain after a close must not appear to the application as file descriptors.

This is existing practice for *mmap*() and *close*(). In addition, there are already mappings present (text, data, stack) that do not have open file descriptors. The text mapping in particular is considered a reference to the file containing the text. The desire was to treat all mappings by the process uniformly. Also, many modern implementations use *mmap*() to implement shared libraries, and it would not be desirable to keep file descriptors for each of the many libraries an application can use. It was felt there were many other existing programs that used this behavior to free a file descriptor, and thus this standard could not forbid it and still claim to be using existing practice.

For implementations that implement memory objects using memory only, memory objects will retain the memory allocated to the file after the last close and will use that same memory on the next open. Note that closing the memory object is not the same as deleting the name, since the memory object is still defined in the memory object name space.

The locks of *fcntl*() do not block any read or write operation, including read or write access to shared memory or mapped files. In addition, implementations that only support shared memory objects should not be required to implement record locks. The reference to *fcntl*() is added to make this point explicitly. The other *fcntl*() commands are useful with shared memory objects.

The size of pages that mapping hardware may be able to support may be a configurable value, or it may change based on hardware implementations. The addition of the {_SC_PAGESIZE} parameter to the *sysconf*() function is provided for determining the mapping page size at runtime.

B.12.2.1 Map Process Addresses to a Memory Object

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, etc., 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 will probably 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 will 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 a *munmap*() before an *mmap*() to the same address range. The standard specifies that the new mappings will replace any existing mappings, following existing practice in this regard.

It is not expected, when the Memory Protection option is supported, that all hardware implementations will be able to support all combinations of permissions at all addresses. When this option is supported, implementations are required to disallow write access to mappings without write permission and to disallow access to mappings without any access permission. Other than these restrictions, implementations may allow access types other than those requested by the application. For example, if the application requests only PROT_WRITE, the implementation may also allow read access. A call to *mmap*() will fail if the implementation cannot support allowing all the access requested by the application. For example, some implementations cannot support a request for both write access and execute access simultaneously. All implementations supporting the Memory Protection option must support requests for no access, read access, write access, and both read and write access. Strictly conforming code must only rely on the required checks. These restrictions allow for portability across a wide range of hardware.

The MAP_FIXED address treatment will likely fail for non-page-aligned values and for certain architecture-dependent address ranges. Conforming implementations cannot count on being able to choose address values for MAP_FIXED without utilizing nonportable, implementation-specific knowledge. Nonetheless, MAP_FIXED is provided as a standard interface conforming to existing practice for utilizing such knowledge when it is available.

Similarly, in order to allow implementations that do not support virtual addresses, support for directly specifying any mapping addresses via MAP_FIXED is not required and thus a conforming application may not count on it.

The MAP_PRIVATE function can be implemented efficiently when memory protection hardware is available. When such hardware is not available, implementations can implement such “mappings” by simply making a real copy of the relevant data into process private memory, though this tends to behave similarly to *read*().

The interface has been defined to allow for many different models of using shared memory. However, all uses are not equally portable across all machine architectures. In particular, the *mmap*() function allows the system as well as the application to specify the address at which to map a specific region of a memory object. The most portable way to use the interface is always to let the system choose the address, specifying **NULL** as the value for the argument *addr* and not to specify MAP_FIXED.

If it is intended that a particular region of a memory object be mapped at the same address in a group of processes (on machines where this is even possible), then MAP_FIXED can be used to pass in the desired mapping address. The system can still be used to choose the desired address if the first such mapping is made without specifying MAP_FIXED, and then the resulting mapping address can be passed to subsequent processes for them to pass in via MAP_FIXED. The availability of a specific address range cannot be guaranteed, in general.

The *mmap*() function can be used to map a region of memory that is larger than the current size of the object. Memory access within the mapping but beyond the current end of the underlying objects may result in SIGBUS signals being sent to the process. The reason for this is that the size of the object can be manipulated by other processes and can change at any moment. The implementation should tell the application that a memory reference is outside the object where this can be detected, otherwise written data may be lost and read data may not reflect actual data in the object.

Note that references beyond the end of the object do not extend the object as the new end cannot be determined precisely by most virtual memory hardware. Instead, the size can be directly manipulated by *ftruncate*().

Process memory locking does apply to shared memory regions, and the MEMLOCK_FUTURE argument to *memlockall*() can be relied upon to cause new shared memory regions to be automatically locked.

Existing implementations of *mmap*() return the value –1 when unsuccessful. Since the casting of this value to type *void** cannot be guaranteed by the C Standard {2} to be distinct from a successful value, this standard defines the symbol MAP_FAILED, which a conforming implementation will not return as the result of a successful call.

B.12.2.2 Unmap Previously Mapped Addresses

The *munmap*() function corresponds to SVR4, just as the *mmap*() function does.

It is possible that an application has applied process memory locking to a region that contains shared memory. If this has occurred, the *munmap*() call will ignore those locks and, if necessary, cause those locks to be removed.

B.12.2.3 Change Memory Protection

There is no additional rationale provided for this subclause.

B.12.2.4 Memory Object Synchronization

The *msync*() function is used to write out data in a mapped region to the permanent storage for the underlying object. The call to *msync*() ensures data integrity of the file.

After the data is written out, any cached data may be invalidated if the MS_INVALIDATE flag was specified. This is useful on systems that do not support read/write consistency.

B.12.3 Shared Memory Functions

Implementations may support the Shared Memory Objects option without supporting a general Memory Mapped Files option. Shared memory objects are named regions of storage that may be independent of the file system and can be mapped into the address space of one or more processes to allow them to share the associated memory.

Requirements

Shared memory is used to share data among several processes, each potentially running at different priority levels, responding to different inputs, or performing separate tasks. Shared memory is not just simply providing common access to data, it is providing the fastest possible communication between the processes. With one memory write operation, a process can pass information to as many processes as have the memory region mapped.

As a result, shared memory provides a mechanism that can be used for all other interprocess communications facilities. It may also be used by an application for implementing more sophisticated mechanisms than semaphores and message queues.

The need for a shared memory interface is obvious for virtual memory systems, where the operating system is directly preventing processes from accessing each other's data. However in unprotected systems, such as those found in some embedded controllers, a shared memory interface is needed to provide a portable mechanism to allocate a region of memory to be shared and then to communicate the address of that region to other processes.

This then, provides the minimum functionality that a shared memory interface must have in order to support realtime applications: to allocate and name a object to be mapped into memory for potential sharing [open() or *shm_open*()], and to make the memory object available within the address space of a process [*mmap*()]. To complete the interface, a mechanism to release the claim of a process on a shared memory object [*munmap*()] is also needed, as well as a mechanism for deleting the name of a sharable object that was previously created [*unlink*() or *shm_unlink*()].

After a mapping has been established, an implementation should not have to provide services to maintain that mapping. All memory writes into that area will appear immediately in the memory mapping of that region by any other processes.

Thus, requirements include

- Support creation of sharable memory objects and the mapping of these objects into the address space of a process.
- Sharable memory objects should be accessed by global names accessible from all processes.
- Support the mapping of specific sections of physical address space (such as a memory mapped device) into the address space of a process. This should not be done by the process specifying the actual address, but again by an implementation-defined global name (such as a special device name) dedicated to this purpose.
- Support the mapping of discrete portions of these memory objects.
- Support for minimum hardware configurations that contain no physical media on which to store shared memory contents permanently.
- The ability to preallocate the entire shared memory region so that minimum hardware configurations without virtual memory support can guarantee contiguous space.

- The maximizing of performance by not requiring functionality that would require implementation interaction above creating the shared memory area and returning the mapping.

Note that the above requirements do not preclude:

- The sharable memory object from being implemented using actual files on an actual file system.
- The global name that is accessible from all processes being restricted to a file system area that is dedicated to handling shared memory.
- An implementation not providing implementation-defined global names for the purpose of physical address mapping.

Shared Memory Objects Usage

If the Shared Memory Objects option is supported, a shared memory object may be created, or opened if it already exists, with the *shm_open*() function. If the shared memory object is created, it has a length of zero. The *ftruncate*() function can be used to set the size of the shared memory object after creation. The *shm_unlink*() function removes the name for a shared memory object created by *shm_open*().

Shared Memory Overview

The shared memory facility defined by this standard usually results in memory locations being added to the address space of the process. The implementation returns the address of the new space to the application by means of a pointer. This works well in languages like C. However, in languages such as FORTRAN, it will not work because these languages do not have pointer types. In the bindings for such a language, either a special COMMON section will need to be defined (which is unlikely), or the binding will have to allow existing structures to be mapped. The implementation will likely have to place restrictions on the size and alignment of such structures or will have to map a suitable region of the address space of the process into the memory object, and thus into other processes. These are issues for that particular language binding. For this standard, however, the practice will not be forbidden, merely undefined.

Two potentially different name spaces are used for naming objects that may be mapped into process address spaces. When the Memory Mapped Files option is supported, files may be accessed via *open*(). When the Shared Memory Objects option is supported, sharable memory objects that might not be files may be accessed via the *shm_open*() function. These options are not mutually exclusive.

Some systems supporting the Shared Memory Objects option may choose to implement the shared memory object name space as part of the file system name space. There are several reasons for this.

- It allows applications to prevent name conflicts by use of the directory structure.
- It uses an existing mechanism for accessing global objects and prevents the creation of a new mechanism for naming global objects.

In such implementations, memory objects can be implemented using regular files, if that is what the implementation chooses. The *shm_open*() function can be implemented as an *open*() call in a fixed directory followed by a call to *fcntl*() to set FD_CLOEXEC. The *shm_unlink*() function can be implemented as an *unlink*() call.

On the other hand, it is also expected that small embedded systems that support the Shared Memory Objects option may wish to implement shared memory without having any file systems present. In this case, the implementations may choose to use a simple string valued name space for shared memory regions. The *shm_open*() function permits either type of implementation.

Some systems have hardware that supports protection of mapped data from certain classes of access and some do not. Systems that supply this functionality can support the Memory Protection option.

Some implementations restrict size, alignment, and protections to be on page-size boundaries. If an implementation has no restrictions on size or alignment, it may specify a 1B page size. Applications on implementations that do support larger pages must be cognizant of the page size since this is the alignment and protection boundary.

Simple embedded implementations may have a 1B page size and only support the Shared Memory Objects option. This provides simple shared memory between processes without requiring mapping hardware.

The standard is silent about how implementations that chose to implement memory objects directly would treat them with standard utilities such as ls because utilities are not within the charter of this standard.

The standard specifically allows a memory object to remain referenced after a close because that is existing practice for the *mmap*() function.

B.12.3.1 Open a Shared Memory Object

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 is used to obtain a descriptor to shared memory object to be mapped.

There is ample precedent for having a file descriptor represent several types of objects. In ISO/IEC 9945-1 : 1990, 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 are returned by *open*() or *creat*() and in some cases returned by alternate functions, such as *pipe*(). The latter technique used by *shm_open*(). Having separate type-specific functions for similar operation was rejected in POSIX.

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 interface 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 a *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 insure 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 section of the standard is silent on the results if the implementation cannot supply the requested file access because of implementation-defined reasons, including hardware ones.

The error codes [EACCES] and [ENOTSUP] are provided to inform the application that the implementation can not complete a request.

[EACCES] indicates for implementation-dependent reasons, probably hardware related, that the implementation cannot comply with a requested mode because it conflicts with another requested mode. A example might be that an application desires to open a memory object two times, mapping different areas with different access modes. If the implementation can not 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-dependent 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-specific 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.

The standard 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.

B.12.3.2 Remove a Shared Memory Object

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 will not be destroyed until the last close and unmap on it have occurred if it was already in use.

B.13 Execution Scheduling

This portion of the rationale presents models, requirements, and standardization issues relevant to priority scheduling.

In an operating system supporting multiple concurrent processes, the system determines the order in which processes execute to meet system-defined goals. For time-sharing systems, the goal is to enhance system throughput and promote fairness; the application is provided little or no control over this sequencing function. While this is acceptable and desirable behavior in a time-sharing system, it is inappropriate in a realtime system; realtime applications must specifically control the execution sequence of their concurrent processes in order to meet externally defined response requirements.

In this standard, the control over process sequencing is provided using a concept of scheduling policies. These policies, described in detail in this section, define the behavior of the system whenever processor resources are to be allocated to competing processes. Only the behavior of the policy is defined; conforming implementations are free to use any mechanism desired to achieve the described behavior.

Models

In an operating system supporting multiple concurrent processes, the system determines the order in which processes execute and might force long-running processes to yield to other processes at certain intervals. Typically, the scheduling code is executed whenever an event occurs that might alter the process to be executed next.

The simplest scheduling strategy is a “first-in, first-out” (FIFO) dispatcher. Whenever a process becomes runnable, it is placed on the end of a ready list. The process at the front of the ready list is executed until it exits or becomes blocked, at which point it is removed from the list. This scheduling technique is also known as “run-to-completion” or “run-to-block.”

A natural extension to this scheduling technique is the assignment of a "nonmigrating priority" to each process. This policy differs from strict FIFO scheduling in only one respect: whenever a process becomes runnable, it is placed at the end of the list of processes runnable at that priority level. When selecting a process to run, the system always selects the first process from the highest priority queue with a runnable process. Thus, when a process becomes unblocked, it will preempt a running process of lower priority without otherwise altering the ready list. Further, if a process elects to alter its priority, it is removed from the ready list and reinserted, using its new priority, according to the policy above.

While the above policy might be considered unfriendly in a time-sharing environment in which multiple users require more balanced resource allocation, it could be ideal in a realtime environment for several reasons. The most important of these is that it is deterministic: the highest-priority process is always run and, among processes of equal priority, the process that has been runnable for the longest time is executed first. Because of this determinism, cooperating processes can implement more complex scheduling simply by altering their priority. For instance, if processes at a single priority were to reschedule themselves at fixed time intervals, a time-slice policy would result.

In a dedicated operating system in which all processes are well-behaved realtime applications, nonmigrating priority scheduling is sufficient. However, many existing implementations provide for more complex scheduling policies.

This part of ISO/IEC 9945 specifies a linear scheduling model. In this model, every process in the system has a priority. The system scheduler always dispatches a process that has the highest (generally the most time-critical) priority among all runnable processes in the system. As long as there is only one such process, the dispatching policy is trivial. When multiple processes of equal priority are eligible to run, they are ordered according to a strict run-to-completion (FIFO) policy.

The priority is represented as a positive integer and is inherited from the parent process. For processes running under a fixed priority scheduling policy the priority is never altered except by an explicit function call.

It was determined arbitrarily that larger integers correspond to "higher priorities."

Certain implementations might impose restrictions on the priority ranges to which processes can be assigned. There also can be restrictions on the set of policies to which processes can be set.

Requirements

Realtime processes require that scheduling be fast and deterministic, and that it guarantees to preempt lower priority processes.

Thus, given the linear scheduling model, realtime processes require that they be run at a priority that is higher than other processes. Within this framework, realtime processes are free to yield execution resources to each other in a completely portable and implementation-independent manner.

As there is a generally perceived requirement for processes at the same priority level to share processor resources more equitably, provisions are made by providing a scheduling policy (that is, SCHED_RR) intended to provide a timeslice-like facility.

NOTE — The following topics assume that low numeric priority implies low scheduling criticality and vice versa.

Rationale for New Interface

Realtime applications need to be able to determine when processes will run in relation to each other. It must be possible to guarantee that a critical process will run whenever it is runnable; that is, whenever it wants to for as long as it needs. SCHED_FIFO satisfies this requirement. Additionally, SCHED_RR was defined to meet a realtime requirement for a well-defined time-sharing policy for processes at the same priority.

It would be possible to use the BSD *setpriority*() and *getpriority*() functions by redefining the meaning of the "nice" parameter according to the scheduling policy currently in use by the process. The System V *nice*() interface was felt to be undesirable for realtime because it specifies an adjustment to the "nice" value, rather than setting it to an explicit value. Realtime applications will usually want to set priority to an explicit value. Also, System V *nice*() does not allow for changing the priority of another process.

With the POSIX.1b interfaces, the traditional "nice" value does not affect the SCHED_FIFO or SCHED_RR scheduling policies. If a "nice" value is supported, it is implementation-defined whether it affects the SCHED_OTHER policy.

An important aspect of the standard is the explicit description of the queuing and preemption rules. It is critical, to achieve deterministic scheduling, that such rules be stated clearly in the standard.

This standard does not address the interaction between priority and swapping. The issues involved with swapping and virtual memory paging are extremely implementation dependent and would be nearly impossible to standardize at this point. The proposed scheduling paradigm, however, fully describes the scheduling behavior of runnable processes, of which one criterion is that the working set be resident in memory. Assuming the existence of a portable interface for locking portions of a process in memory, paging behavior need not affect the scheduling of realtime processes.

This standard also does not address the priorities of "system" processes. In general, these processes should always execute in low-priority ranges to avoid conflict with other realtime processes. Implementations should document the priority ranges in which system processes run.

The default scheduling policy is not defined. The effect of I/O interrupts and other system processing activities is not defined. The temporary lending of priority from one process to another (such as for the purposes of affecting freeing resources) by the system is not addressed. Preemption of resources is not addressed. Restrictions on the ability of a process to affect other processes beyond a certain level (influence levels) is not addressed.

The rationale used to justify the simple time-quantum scheduler is that it is common practice to depend upon this type of scheduling to assure "fair" distribution of processor resources among portions of the application that must interoperate in a serial fashion. Note that the standard is silent with respect to the setting of this time quantum, or whether it is a systemwide value or a per-process value, although it appears that the prevailing realtime practice is for it to be a systemwide value.

In a system with N processes at a given priority, all processor-bound, in which the time quantum is equal for all processes at a specific priority level, the following assumptions are made of such a scheduling policy:

1) A time quantum Q exists and the current process will own control of the processor for at least a duration of Q and will have the processor for a duration of Q.
2) The Nth process at that priority will control a processor within a duration of $(N-1) \times Q$.

These assumptions are necessary to provide equal access to the processor and bounded response from the application.

The assumptions hold for the described scheduling policy only if no system overhead, such as interrupt servicing, is present. If the interrupt servicing load is nonzero, then one of the two assumptions becomes fallacious, based upon how Q is measured by the system.

If Q is measured by clock time, then the assumption that the process obtains a duration Q processor time is false if interrupt overhead exists. Indeed, a scenario can be constructed with N processes in which a single process undergoes complete processor starvation if a peripheral device, such as an analog-to-digital converter, generates significant interrupt activity periodically with a period of $N \times Q$.

If Q is measured as actual processor time, then the assumption that the Nth process runs in within the duration $(N-1) \times Q$ is false.

It should be noted that SCHED_FIFO suffers from interrupt-based delay as well. However, for SCHED_FIFO, the implied response of the system is "as soon as possible," so that the interrupt load for this case is a vendor selection and not a compliance issue.

With this in mind, it is necessary either to complete the definition by including bounds on the interrupt load, or to modify the assumptions that can be made about the scheduling policy.

Since the motivation of inclusion of the policy is common usage, and since current applications do not enjoy the luxury of bounded interrupt load, item (2) above is sufficient to express existing application needs and is less restrictive in the standard definition. No difference in interface is necessary.

In an implementation in which the time quantum is equal for all processes at a specific priority, our assumptions can then be restated as:

— A time quantum Q exists, and a processor-bound process will be rescheduled after a duration of, at most, Q. Time quantum Q may be defined in either wall clock time or execution ti me.
— In general, the Nth process of a priority level should wait no longer than $(N–l) \times Q$ time to execute, assuming no processes exist at higher priority levels.
— No process should wait indefinitely.

For implementations supporting per-process time quanta, these assumptions can be readily extended.

B.13.1 Thread Scheduling

Scheduling Implementation Models

The following scheduling implementation models are presented in terms of threads and "kernel entities." This is to simplify exposition of the models, and it does not imply that an implementation actually has an identifiable "kernel entity."

A kernel entity is not defined beyond the fact that it has scheduling attributes that are used to resolve contention with other kernel entities for execution resources. A kernel entity may be thought of as an envelope that holds a thread or a separate kernel thread. It is not a conventional process, although it shares with the process the attribute that it has a single thread of control; it does not necessarily imply an address space, open files, etc. It is better thought of as a primitive facility upon which conventional processes and threads may be %constructed.

— *System Thread Scheduling Model*: This model consists of one thread per kernel entity. The kernel entity is solely responsible for scheduling thread execution on one or more processors. This model schedules all threads against all other threads in the system using the scheduling attributes of the thread.
— *Process Scheduling Model*: A generalized process scheduling model consists of two levels of scheduling. A threads library creates a pool of kernel entities, as required, and schedules threads to run on them using the scheduling attributes of the threads. Typically, the size of the pool is a function of the simultaneously runnable threads, not the total number of threads. The kernel then schedules the kernel entities onto processors according to their scheduling attributes, which are managed by the threads library. This set model potentially allows a wide range of mappings between threads and kernel entities.

System and Process Scheduling Model Performance

There are a number of important implications on the performance of applications using these scheduling models. The process scheduling model potentially provides lower overhead for making scheduling decisions, since there is no need to access kernel-level information or functions and the set of schedulable entities is smaller (only the threads within the process).

On the other hand, since the kernel is also making scheduling decisions regarding the system resources under its control (e.g., CPU(s), I/O devices, memory), decisions that do not take thread scheduling parameters into account can result in indeterminate delays for realtime application threads, causing them to miss maximum response time limits.

Rate Monotonic Scheduling

Rate Monotonic Scheduling was considered, but rejected for standardization in the context of threads. It and a sporadic server policy are being considered in the context of further POSIX standardization efforts.

Scheduling Options

In this standard, the basic thread scheduling functions are defined under the {_POSIX_THREADS} option so that they are required of all threads implementations. However, there are no specific scheduling policies required by this option to allow for conforming thread implementations that are not targeted to realtime applications.

Specific standard scheduling policies are defined to be under the {_POSIX_THREAD_PRIORITY_SCHEDULING} option, and they are specifically designed to support realtime applications by providing predictable resource sharing sequences. The name of this option was chosen to emphasize that this functionality is defined as appropriate for realtime applications that require simple priority-based scheduling.

It is recognized that these policies are not necessarily satisfactory for some multiprocessor implementations, and work is ongoing to address a wider range of scheduling behaviors. The interfaces have been chosen to create abundant opportunity for future scheduling policies to be implemented and standardized based on this interface. In order to standardize a new scheduling policy, all that is required (from the standpoint of thread scheduling attributes) is to define a new policy name, new members of the thread attributes object, and functions to set these members when the scheduling policy is equal to the new value.

B.13.1.1 Scheduling Contention Scope

In order to accommodate the requirement for realtime response, each thread has a scheduling contention scope attribute. Threads with a system scheduling contention scope have to be scheduled with respect to all other threads in the system. These threads are usually bound to a single kernel entity that reflects their scheduling attributes and are directly scheduled by the kernel.

Threads with a process scheduling contention scope need be scheduled only with respect to the other threads in the process. These threads may be scheduled within the process onto a pool of kernel entities. The implementation is also free to bind these threads directly to kernel entities and let them be scheduled by the kernel. Process scheduling contention scope allows the implementation the most flexibility and is the default if both contention scopes are supported and none is specified.

Thus, the choice by implementors to provide one or the other (or both) of these scheduling models is driven by the need of their supported application domains for worst-case (i.e., realtime) response, or average-case (nonrealtime) response.

B.13.1.2 Scheduling Allocation Domain

The SCHED_FIFO and SCHED_RR scheduling policies take on different characteristics on a multiprocessor. Other scheduling policies are also subject to changed behavior when executed on a multiprocessor. The concept of scheduling allocation domain determines the set of processors on which the threads of an application may run. By considering the application's processor scheduling allocation domain for its threads, scheduling policies can be defined in terms of their behavior for varying processor scheduling allocation domain values. It is conceivable that not all scheduling allocation domain sizes make sense for all scheduling policies on all implementations. The concept of scheduling allocation domain, however, is a useful tool for the description of multiprocessor scheduling policies.

The “process control” approach to scheduling (See {B72}) obtains significant performance advantages from dynamic scheduling allocation domain sizes when it is applicable.

Non-Uniform Memory Access (NUMA) multiprocessors (for instance, BBN Butterfly, Stanford DASH) may use a system scheduling structure that involves reassignment of threads among scheduling allocation domains. In NUMA machines, a natural model of scheduling is to match scheduling allocation domains to clusters of processors. Load balancing in such an environment requires changing the scheduling allocation domain to which a thread is assigned.

B.13.1.3 Scheduling Documentation

Implementation-provided scheduling policies need to be completely documented in order to be useful. This documentation includes a description of the attributes required for the policy, the scheduling interaction of threads running under this policy and all other supported policies, and the effects of all possible values for processor scheduling allocation domain. Note that for the implementor wishing to be minimally compliant, it is (minimally) acceptable to define the behavior as undefined.

B.13.2 Thread Scheduling Functions

B.13.2.1 Thread Creation Scheduling Attributes

Scheduling Contention Scope Attribute

The scheduling contention scope defines how threads compete for resources. Within this standard, scheduling contention scope is used to describe only how threads are scheduled in relation to one another in the system. That is, either they are scheduled against all other threads in the system (“system scope”) or only against those threads in the process (“process scope”). In fact, scheduling contention scope may apply to additional resources, including virtual timers and profiling, which are not currently considered by this standard.

Mixed Scopes

If only one scheduling contention scope is supported, the scheduling decision is straightforward. To perform the processor scheduling decision in a mixed scope environment, it is necessary to map the scheduling attributes of the thread with process-wide contention scope to the same attribute space as the thread with system-wide contention scope.

Since a conforming implementation has to support one and may support both scopes, it is useful to discuss the effects of such choices with respect to example applications. If an implementation supports both scopes, mixing scopes provides a means of better managing system-level (that is, kernel level) and library-level resources. In general, threads with system scope will require the resources of a separate kernel entity in order to guarantee the scheduling semantics. On the other hand, threads with process scope can share the resources of a kernel entity while maintaining the scheduling semantics.

The application is free to create threads with dedicated kernel resources, and other threads that multiplex kernel resources. Consider the example of a window server. The server allocates two threads per widget: one thread manages the widget user interface (including drawing), while the other thread takes any required application action. This allows the widget to be “active” while the application is computing. A screen image may be built from thousands of widgets. If each of these threads had been created with system scope, then most of the kernel level resources might be wasted, since only a few widgets are active at any one time. In addition, mixed scope is particularly useful in a window server where one thread with high priority and system scope handles the mouse so that it tracks well. As another example, consider a database server. For each of the hundreds or thousands of clients supported by a large server, an equivalent number of threads will have to be created. If each of these threads were system, the consequences would be the same as for the window server example above. However, the server could be constructed so that actual retrieval of data is done by several dedicated threads. Dedicated threads that do work for all clients frequently justify the added expense of system scope. If it were not permissible to mix system and process threads in the same process, this type of solution would not be possible.

B.13.2.2 Dynamic Thread Scheduling Parameters Access

In many time-constrained applications, there is no need to change the scheduling attributes dynamically during thread or process execution, since the general use of these attributes is to reflect directly the time constraints of the application. Since these time constraints are generally imposed to meet higher-level system requirements, such as accuracy or availability, they frequently should remain unchanged during application execution.

However, there are important situations in which the scheduling attributes should be changed. Generally, this will occur when external environmental conditions exist in which the time constraints change. Consider, for example, a space vehicle major mode change, such as the change from ascent to descent mode, or the change from the space environment to the atmospheric environment. In such cases, the frequency with which many of the sensors or acutators need to be read or written will change, which will necessitate a priority change. In other cases, even the existence of a time constraint might be temporary, necessitating not just a priority change, but also a policy change for ongoing threads or processes. For this reason, it is critical that the interface should provide functions to change the scheduling parameters dynamically, but, as with many of the other real-time functions, it is important that applications use them properly to avoid the possibility of unnecessarily degrading performance.

In providing functions for dynamically changing the scheduling behavior of threads, there were two options: provide functions to get and set the individual scheduling parameters of threads, or provide a single interface to get and set all the scheduling parameters for a given thread simultaneously. Both approaches have merit. Access functions for individual parameters allow simpler control of thread scheduling for simple thread scheduling parameters. However, a single function for setting all the parameters for a given scheduling policy is required when first setting that scheduling policy. Since the single all-encompassing functions are required, it was decided to leave the interface as minimal as possible. Note that simpler functions (such as *pthread_setprio*() for threads running under the priority-based schedulers) can be easily defined in terms of the all-encompassing functions.

If the function *pthread_setschedparam*() executes successfully, it will have set all of the scheduling parameter values indicated in *param*; otherwise, none of the scheduling parameters will have been modified. This is necessary to ensure that the scheduling of this and all other threads continues to be consistent in the presence of an erroneous scheduling parameter.

The [EPERM] value is included in the list of possible *pthread_setschedparam*() error returns as a reflection of the fact that the ability to change scheduling parameters increases risks to the implementation and application performance if the scheduling parameters are changed improperly. For this reason, and based on some existing practice, it was felt that some implementations would probably choose to define specific permissions for changing either a thread's own or another thread's scheduling parameters. The standard does not include portable methods for setting or retrieving permissions, so any such use of permissions is completely unspecified.

B.13.2.3 Mutex Initialization Scheduling Attributes

In a priority-driven environment, a direct use of traditional primitives like mutexes and condition variables can lead to unbounded priority inversion, where a higher priority thread can be blocked by a lower priority thread, or set of threads, for an unbounded duration of time. As a result, it becomes impossible to guarantee thread deadlines. Priority inversion can be bounded and minimized by the use of priority inheritance protocols. This allows thread deadlines to be guaranteed even in the presence of synchronization requirements.

Two useful but simple members of the family of priority inheritance protocols are the basic priority inheritance protocol and the priority ceiling protocol emulation. Under the Basic Priority Inheritance protocol (governed by the {_POSIX_THREAD_PRIO_INHERIT} option), a thread that is blocking higher priority threads executes at the priority of the highest priority thread that it blocks. This simple mechanism allows priority inversion to be bounded by the duration of critical sections and makes timing analysis possible.

Under the Priority Ceiling Protocol Emulation protocol (governed by the {_POSIX_THREAD_PRIO_PROTECT} option), each mutex has a priority ceiling, usually defined as the priority of the highest priority thread that can lock the mutex. When a thread is executing inside critical sections, its priority is unconditionally increased to the highest of the priority ceilings of all the mutexes owned by the thread. This protocol has two very desirable properties in uniprocessor systems. First, a thread can be blocked by a lower priority thread for at most the duration of one single critical section. Furthermore, when the protocol is correctly used in a single processor, and if threads do not become blocked while owning mutexes, mutual deadlocks are prevented.

The priority ceiling emulation can be extended to multiple processor environments, in which case the values of the priority ceilings will be assigned depending on the kind of mutex that is being used: local to only one processor, or global, shared by several processors. Local priority ceilings will be assigned the usual way, equal to the priority of the highest priority thread that may lock that mutex. Global priority ceilings will usually be assigned a priority level higher than all the priorities assigned to any of the threads that reside in the involved processors to avoid the effect called remote blocking.

B.13.2.4 Change the Priority Ceiling of a Mutex

In order for the priority protect protocol to exhibit its desired properties of bounding priority inversion and avoidance of deadlock, it is critical that the ceiling priority of a mutex be the same as the priority of the highest thread that can ever hold it, or higher. Thus, if the priorities of the threads using such mutexes never change dynamically, there is no need ever to change the priority ceiling of a mutex.

However, if a major system mode change results in an altered response time requirement for one or more application threads, their priority has to change to reflect it. It will occasionally be the case that the priority ceilings of mutexes held also need to change. While changing priority ceilings should generally be avoided, it is important that the standard provide these interfaces for those cases in which it is necessary.

B.14 Clocks and Timers

Clocks

ISO/IEC 9945-1 :1990 and the C Standard {2} both define functions for obtaining System Time. Implicit behind these functions is a mechanism for measuring passage of time. This specification makes this mechanism explicit and calls it a clock. The CLOCK_REALTIME clock required by the standard is a higher resolution version of the clock that maintains ISO/IEC 9945-1 :1990 System Time. This is a “systemwide” clock in that it is visible to all processes and, were it possible for multiple processes to all read the clock at the same time, they would see the same value.

An extensible interface was defined, with the ability for implementations to define additional clocks. This was done because of the observation that many realtime platforms support multiple clocks, and it was desired to fit this model within the standard interface. But implementation-defined clocks need not represent actual hardware devices, nor are they necessarily systemwide.

Timers

Two timer types are required for a system to support realtime applications.

— *One-shot*
  A one-shot timer is a timer that is armed with an initial expiration time, either relative to the current time or at an absolute time (based on some timing base, such as time in seconds and nanoseconds since the Epoch). The timer expires once and then is disarmed. With the specified facilities, this is accomplished by setting the *it_value* member of the *value* argument to the desired expiration time and the *it_interval* member to zero.
— *Periodic*
  A periodic timer is a timer that is armed with an initial expiration time, again either relative or absolute, and a repetition interval. When the initial expiration occurs, the timer is reloaded with the repetition interval and

continues counting. With the specified facilities, this is accomplished by setting the *it_value* member of the *value* argument to the desired initial expiration time and the *it_interval* member to the desired repetition interval.

For both of these types of timers, the time of the initial timer expiration can be specified in two ways:

— Relative (to the current time) or
— Absolute

Examples of Using Realtime Timers

In the diagrams below, "S" indicates a program schedule, "R" shows a schedule method request, and "E" suggests an internal operating system event.

— *Periodic timer—data logging*
During an experiment, it might be necessary to log realtime data periodically to an internal buffer or to a mass storage device. With a periodic scheduling method, a logging module can be started automatically at fixed time intervals to log the data.
Program schedule is requested every 10 s.

```
    R         S         S         S         S         S
----+----+----+----+----+----+----+----+----+----+----+--->
    5   10   15   20   25   30   35   40   45   50   55
```

Time (in seconds)

To achieve this type of scheduling using the specified facilities, one would allocate a per-process timer based on clock ID CLOCK_REALTIME. Then the timer would be armed via a call to *timer_settime*() with the TIMER_ABSTIME flag reset, and with an initial expiration value and a repetition interval of 10 s.

— *One-shot timer (relative time)—device initialization*
In an emission test environment, large sample bags are used to capture the exhaust from a vehicle. The exhaust is purged from these bags before each and every test. With a one-shot timer, a module could initiate the purge function and then suspend itself for a predetermined period of time while the sample bags are prepared.
Program schedule requested 20 s after call is issued.

```
    R                   S
----+----+----+----+----+----+----+----+----+----+----+--->
    5   10   15   20   25   30   35   40   45   50   55
```

Time (in seconds)

To achieve this type of scheduling using the specified facilities, one would allocate a per-process timer based on clock ID CLOCK_REALTIME. Then the timer would be armed via a call to *timer_settime*() with the TIMER_ABSTIME flag reset, and with an initial expiration value of 20 s and a repetition interval of zero.
Note that if the program wishes merely to suspend itself for the specified interval, it could more easily use *nanosleep*().

— *One-shot timer (absolute time)—data transmission*
The results from an experiment are often moved to a different system within a network for postprocessing or archiving. With an absolute one-shot timer, a module that moves data from a test-cell computer to a host computer can be automatically scheduled on a daily basis.
Program schedule requested for 2:30 a.m.

```
         R                                     S
-----+-----+-----+-----+-----+-----+-----+-----+-----+----->
   23:00 23:30 24:00 00:30 01:00 01:30 02:00 02:30 03:00
```

Time of Day

To achieve this type of scheduling using the specified facilities, one would allocate a per-process timer based on clock ID CLOCK_REALTIME. Then the timer would be armed via a call to *timer_settime*() with the TIMER_ABSTIME flag set, and an initial expiration value equal to 2:30 a.m. of the next day.

— *Periodic timer (relative time)—signal stabilization*
Some measurement devices, such as emission analyzers, do not respond instantaneously to an introduced sample. With a periodic timer with a relative initial expiration time, a module that introduces a sample and records the average response could suspend itself for a predetermined period of time while the signal is stabilized and then sample at a fixed rate.
Program schedule requested 15 s after call is issued and every 2 s thereafter.

```
   R              S S S S S S S S S S S S S S S S S S S S
----+----+----+----+----+----+----+----+----+----+----+--->
    5   10   15   20   25   30   35   40   45   50   55
```

Time (in seconds)

To achieve this type of scheduling using the specified facilities, one would allocate a per-process timer based on clock ID CLOCK_REALTIME. Then the timer would be armed via a call to *timer_settime*() with TIMER_ABSTIME flag reset, and with an initial expiration value of 15 s and a repetition interval of 2 s.

— *Periodic timer (absolute time)—work-shift related processing*
Resource utilization data is useful when time to perform experiments is being scheduled at a facility. With a periodic timer with an absolute initial expiration time, a module can be scheduled at the beginning of a work shift to gather resource utilization data throughout the shift. This data can be used to allocate resources effectively to minimize bottlenecks and delays and maximize facility throughput.
Program schedule requested for 2:00 a.m. and every 15 min thereafter.

```
         R                               S  S  S  S  S  S
-----+-----+-----+-----+-----+-----+-----+-----+-----+----->
   23:00 23:30 24:00 00:30 01:00 01:30 02:00 02:30 03:00
```

Time of Day

To achieve this type of scheduling using the specified facilities, one would allocate a per-process timer based on clock ID CLOCK_REALTIME. Then the timer would be armed via a call to *timer_settime*() with TIMER_ABSTIME flag set, and with an initial expiration value equal to 2:00 a.m. and a repetition interval equal to 15 min.

Relationship of Timers to Clocks

The relationship between clocks and timers armed with an absolute time is straightforward: a timer expiration signal is requested when the associated clock reaches or exceeds the specified time. The relationship between clocks and timers armed with a relative time (an interval) is less obvious, but not unintuitive. In this case, a timer expiration signal is requested when the specified interval, *as measured by the associated clock*, has passed. For the required CLOCK_REALTIME clock, this allows timer expiration signals to be requested at specified "wall clock" times (absolute), or when a specified interval of "realtime" has passed (relative). For an implementation-defined clock—say, a process virtual time clock—timer expirations could be requested when the process has used a specified total amount of virtual time (absolute), or when it has used a specified *additional* amount of virtual time (relative).

The interfaces also allow flexibility in the implementation of the functions. For example, an implementation could convert all absolute times to intervals by subtracting the clock value at the time of the call from the requested expiration time and "counting down" at the supported resolution. Or it could convert all relative times to absolute expiration time by adding in the clock value at the time of the call and comparing the clock value to the expiration time at the supported resolution. Or it might even choose to maintain absolute times as absolute and compare them to the clock value at the supported resolution for absolute timers, and maintain relative times as intervals and count them down at the resolution supported for relative timers. The choice will be driven by efficiency considerations and the underlying hardware or software clock implementation.

B.14.1 Data Definitions for Clocks and Timers

The standard uses a time representation capable of supporting nanosecond resolution timers for the following reasons:

- To enable the standard to represent those computer systems already using nanosecond or submicrosecond resolution clocks.
- To accommodate those per-process timers that might need nanoseconds to specify an absolute value of systemwide clocks even though the resolution of the per-process timer may only be milliseconds, or vice versa.
- Because the number of nanoseconds in a second can be represented in 32 b.

Time values are represented in the *timespec* structure. The *tv_sec* member is of type *time_t* so that this member is compatible with time values used by ISO/IEC 9945-1 : 1990 functions and the C Standard {2}. The *tv_nsec* member is a signed long in order to simplify and clarify code that decrements or finds differences of time values. Note that because i billion (number of nanoseconds per second) is less than half of the value representable by a signed 32 b value, it is always possible to add two valid fractional seconds represented as integral nanoseconds without overflowing the signed 32 b value.

A maximum allowable resolution for the CLOCK_REALTIME clock of 20 ms (1/50 s) was chosen to allow line frequency clocks in European countries to be conforming. 60 Hz clocks in the U.S. will also be conforming, as will finer granularity clocks, although a Strictly Conforming Application cannot assume a granularity of less than 20 ms (1/50 s).

The minimum allowable maximum time allowed for the CLOCK_REALTIME clock and the function *nanosleep*(), and timers created with *clock_id* = CLOCK_REALTIME, is determined by the fact that the *tv_sec* member is of type *time_t*. Earlier development of this part of ISO/IEC 9945 provided for different maximums for timers and *nanosleep*(). This provision was removed when the resolution functions were removed because existing practice does not specify a maximum, but accepts any interval that can be described by a *time_t*.

The standard specifies that timer expirations shall not be delivered early nor shall *nanosleep*() return early due to quantization error. ISO/IEC 9945-1 :1990 discusses the various implementations of *alarm*() in the rationale and states that implementations that do not allow alarm signals to occur early are the most appropriate, but refrained from mandating this behavior. Because of the importance of predictability to realtime applications, this standard takes a stronger stance.

The developers of this part of ISO/IEC 9945 considered using a time representation that differs from POSIX. lb in the second 32 b of the 64-b value. Whereas POSIX.lb defines this field as a fractional second in nanoseconds, the other methodology defines this as a binary fraction of one second, with the radix point assumed before the most significant bit.

POSIX.lb is a software, source-level standard and most of the benefits of the alternate representation are enjoyed by hardware implementations of clocks and algorithms. It was felt that mandating this format for POSIX.lb clocks and timers would unnecessarily burden the application writer with writing, possibly non-portable, multiple precision arithmetic packages to perform conversion between binary fractions and integral units such as nanoseconds, milliseconds, etc.

B.14.2 Clock and Timer Functions

B.14.2.1 Clocks

The need for clock drift rate adjustment was pointed out for two reasons:

1) To provide a mechanism for writing portable clock synchronization algorithms for synchronizing clocks in distributed or cooperating realtime systems, and
2) To provide a way to set a clock while ensuring that time, as measured by the clock, is a monotonically increasing quantity. This is analogous to the capability provided by the BSD *adjtime*() function.

Interfaces were defined to support the synchronization of clocks with external time or between systems by using *clock_settime*() for gross adjustments in time, and by using *clock_setdrift*() to adjust the time precisely. When a clock needs to be set back in time, a negative drift can be used to slow down the clock so that it converges over time to the correct value without invalidating programs that expect time to be monotonically increasing. It also minimizes the impact of changing the time base on armed per-process timers based on that clock.

There is no equivalent to the BSD *adjtime*() function in this part of ISO/IEC 9945. The *adjtime*() function could be implemented on the clock-drift functions described in this rationale. The virtual and profiling interval timing functions of BSD are not in this part of ISO/IEC 9945 , but these could be implemented as extensions using new *clock_id* values.

B.14.2.2 Create a Per-Process Timer

Periodic Timer Overrun and Resource Allocation

The specified timer facilities may deliver Realtime Signals (that is, queued signals) on implementations that support this option. Because 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 indeterminate 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 ISO/IEC 9945-1 : 1990, 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 interfaces, 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. Because timer_ids are dynamically allocated by the *timer_create*() function, they can be used for neither of these functions without additional application overhead in the signal handler—for example, to search an array of saved timer IDs to associate the ID with a constant or application data structure.

The revised text makes consistent the semantics of the members of the *sigevent* structure.

B.14.2.3 Delete a Per-Process Timer

There is no additional rationale provided for this subclause.

B.14.2.4 Per-Process Timers

The *clock_settime*(), *timer_settime*() and *nanosleep*() functions are defined to truncate specified time values down to the resolution supported by the implementation. Values are truncated when set because this appears to be existing practice, and it does not seem reasonable to require an error in this case. Note that this is symmetric with the truncation that occurs when reading the time via *clock_gettime*() or *timer_gettime*() at a time that is not an integral multiple of the clock or timer resolution.

The specification defines interfaces 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 run-time application issue.

B.14.2.5 High Resolution Sleep

It is common to suspend execution of a process for an interval in order to poll the status of a noninterrupting interface. A large number of actual needs can be met with a simple extension to *sleep*() that provides finer resolution.

In ISO/IEC 9945-1 : 1990 and SVR4, it is possible to implement such a routine, but the frequency of wakeup is limited by the resolution of the *alarm*() and *sleep*() functions. In BSD4.3, it is possible to write such a routine using no static storage and reserving no system facilities. Although it is possible to write a function with similar functionality to *sleep*() using the remainder of the timers interface, such a function will require the use of signals and the reservation of some signal number. This standard requires that *nanosleep*() be nonintrusive of the signals interface.

The *nanosleep*() function returns 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.

B.15 Message Passing

This section provides the rationale for the definition of the message passing interface in this standard. This is presented in terms of the objectives, models, and requirements imposed upon this interface.

Objectives

Many applications, including both realtime and database applications, require a means of passing arbitrary amounts of data between cooperating processes comprising the overall application on one or more processors. Many conventional interfaces for interprocess communication are insufficient for realtime applications in that efficient and deterministic data passing methods cannot be implemented. This has prompted the definition of message passing interfaces providing these facilities:

- Open a message queue.
- Send a message to a message queue.
- Receive a message from a queue, either synchronously or asynchronously.
- Alter message queue attributes for flow and resource control.

It is assumed that an application may consist of multiple cooperating processes and that these processes may wish to communicate and coordinate their activities. The message passing facility described in this standard allows processes to communicate through system wide queues. These message queues are accessed through names that may be pathnames. A message queue can be opened for use by multiple sending and/or multiple receiving processes.

Background on Embedded Applications

Interprocess communication utilizing message passing is a key facility for the construction of deterministic, high-performance realtime applications. The facility is present in all realtime systems and is the framework upon which the application is constructed. The performance of the facility is usually a direct indication of the performance of the resulting application.

Realtime applications, especially for embedded systems, are typically designed around the performance constraints imposed by the message passing mechanisms. Applications for embedded systems are typically very tightly constrained. Application writers expect to design and control the entire system. In order to minimize system costs, the writer will attempt to use all resources to their utmost and minimize the requirement to add additional memory or processors.

The embedded applications usually share address spaces and only a simple message passing mechanism is required. The application can readily access common data incurring only mutual exclusion overheads. The models desired are the simplest possible with the application building higher level facilities only when needed.

Requirements

The following requirements determined the features of the message passing facilities defined in this part of ISO/IEC 9945.

- *Naming of message queues*
  The mechanism for gaining access to a message queue is a pathname evaluated in a context that is allowed to be a file system name space, or it can be independent of any file system. This is a specific attempt to allow implementations based on either method in order to address both embedded systems and to also allow implementation in larger systems.
  The interface of *mq_open*() is defined to allow but not require the access control and name conflicts resulting from utilizing a file system for name resolution. All required behavior is specified for the access control case. Yet a conforming implementation, such as an embedded system kernel, may define that there are no distinctions between users and may define that all process have all access privileges.
- *Embedded system naming*
  Embedded systems need to be able to utilize independent name spaces for accessing the various system objects. They typically do not have a file system, precluding its utilization as a common name resolution mechanism. The modularity of an embedded system limits the connections between separate mechanisms that can be allowed.

Embedded systems typically do not have any access protection. Since the system does not support the mixing of applications from different areas, and usually does not even have the concept of an authorization entity, access control is not useful.

- *Large system naming*
  On systems with more functionality, the name resolution must support the ability to use the file system as the name resolution mechanism/object storage medium and to have control over access to the objects. Utilizing the pathname space can result in further errors when the names conflict with other objects.
- *Fixed size of messages*
  The interfaces impose a fixed upper bound on the size of messages that can be sent to a specific message queue. The size is set on an individual queue basis and cannot be changed dynamically.
  The purpose of the fixed size is to increase the ability of the system to optimize the implementation of *mq_send*() and *mq_receive*(). With fixed sizes of messages and fixed numbers of messages, specific message blocks can be pre-allocated. This eliminates a significant amount of checking for errors and boundary conditions. Additionally, an implementation can optimize data copying to maximize performance. Finally, with a restricted range of message sizes, an implementation is better able to provide deterministic operations.
- *Prioritization of messages*
  Message prioritization allows the application to determine the order in which messages are received. Prioritization of messages is a key facility that is provided by most realtime kernels and is heavily utilized by the applications. The major purpose of having priorities in message queues is to avoid priority inversions in the message system, where a high-priority message is delayed behind one or more lower-priority messages. It has been observed that a significant problem with Ada rendezvous is that it queues tasks in strict FIFO order, ignoring priorities. This allows the applications to be designed so that they do not need to be interrupted in order to change the flow of control when exceptional conditions occur. The prioritization does add additional overhead to the message operations in those cases it is actually used but a clever implementation can optimize for the FIFO case to make that more efficient.
- *Asynchronous notification*
  The interface supports the ability to have a task asynchronously notified of the availability of a message on the queue. The purpose of this facility is to allow the task to perform other functions and yet still be notified that a message has become available on the queue.
  To understand the requirement for this function, it is useful to understand two models of application design: a single task performing multiple functions and multiple tasks performing a single function. Each of these models has advantages.
  Asynchronous notification is required to build the model of a single task performing multiple operations. This model typically results from either the expectation that interruption is less expensive than utilizing a separate task or from the growth of the application to include additional functions.

B.15.1 Data Definitions for Message Queues

B.15.1.1 Data Structures

There is no specific rationale for this subclause.

B.15.2 Message Passing Functions

There are several facilities that are not supported by the specified interfaces. The primary reason for this is that they adversely impact performance and are not present in the realtime kernels examined.

The following facilities were explicitly excluded from the interfaces. Some of the rationale behind the exclusion is provided.

- *Buffer management*
  Buffer management is a controversial area. With additional information from the user, the system can make significant performance improvements by eliminating the copying if the messages. But this is only useful if the message buffer is a relatively large, natural allocation unit, such as a page, and the destination is local.

For the embedded applications, only the “local” requirement is met. The assumptions are that the messages are small.

— *Multiple waiting*

The ability to wait on multiple message queues simultaneously has been indicated as a highly desirable facility but adds significant implementation complexity and is not present in most of the realtime kernels.
The implementation of waiting for messages on multiple queues tends to be complex. There is the necessity of placing an indication in each of the queues that a specific process wishes to receive a message. It is then also required that after the first message is received, that no more messages are assigned to the process. Consideration of the possibility of multiple processes waiting for messages on multiple disjoint sets of queues gives insight into the possible implementation complexity.

B.15.2.1 Open a Message Queue

The revised text clarifies the creation of a message queue description and associates the O_NONBLOCK flag with the message queue description.

B.15.2.2 Close a Message Queue

There is no specific rationale for this subclause.

B.15.2.3 Remove a Message Queue

There is no specific rationale for this subclause.

B.15.2.4 Send a Message to a Message Queue

The value of the symbol {MQ_PRIO_MAX} limits the number of priority levels supported by the application. The revised text requires that message priorities range from 0 to {MQ_PRIO_MAX}–1.

B.15.2.5 Receive a Message From a Message Queue

There is no specific rationale for this subclause.

B.15.2.6 Notify Process that a Message is Available on a Queue

There is no specific rationale for this subclause.

B.15.2.7 Set Message Queue Attributes

The revised text clarifies the creation of a message queue description and associates the O_NONBLOCK flag with the message queue description.

B.15.2.8 Get Message Queue Attributes

The revised text clarifies the creation of a message queue description and associates the O_NONBLOCK flag with the message queue description.

B.16 Thread Management

B.16.1 Threads

Threads will normally be more expensive than subroutines (or functions, routines, etc.) if specialized hardware support is not provided. Nevertheless, threads should be sufficiently efficient to encourage their use as a medium- to fine-grained structuring mechanism for parallelism in an application. Structuring an application using threads then allows it to take immediate advantage of any underlying parallelism available in the host environment. This means implementors are encouraged to optimize for fast execution at the possible expense of efficient utilization of storage. For example, a common thread creation technique is to cache appropriate thread data structures. That is, rather than releasing system resources, the implementation retains these resources and reuses them when the program next asks to create a new thread. If this reuse of thread resources is to be possible, there has to be very little unique state associated with each thread, because any such state has to be reset when the thread is reused.

B.16.2 Thread Functions

B.16.2.1 Thread Creation Attributes

Attributes objects are provided for threads, mutexes, and condition variables as a mechanism to support probable future standardization in these areas without requiring that the interface itself be changed. Attributes objects provide clean isolation of the configurable aspects of threads. For example, “stack size” is an important attribute of a thread, but it cannot be expressed portably. When porting a threaded program, stack sizes often need to be adjusted. The use of attributes objects can help by allowing the changes to be isolated in a single place, rather than being spread across every instance of thread creation.

Attributes objects can be used to set up “classes” of threads with similar attributes, for example, “threads with large stacks and high priority” or “threads with minimal stacks.” These classes can be defined in a single place and then referenced wherever threads need to be created. Changes to “class” decisions become straightforward, and detailed analysis of each *pthread_create*() call is not required.

The attributes objects are defined as opaque types as an aid to extensibility. If these objects had been specified as structures, adding new attributes would force recompilation of all multithreaded programs when the attributes objects are extended; this might not be possible if different program components were supplied by different vendors.

Additionally, opaque attributes objects present opportunities for improving performance. Argument validity can be checked once when attributes are set, rather than each time a thread is created. Implementations will often need to cache kernel objects that are expensive to create. Opaque attributes objects provide an efficient mechanism to detect when cached objects become invalid due to attribute changes.

Because assignment is not necessarily defined on a given opaque type, implementation-dependent default values cannot be defined in a portable way. The solution to this problem is to allow attribute objects to be initialized dynamically by attributes object initialization functions, so that default values can be supplied automatically by the implementation.

The following proposal was provided as a suggested alternative to the supplied attributes:

1) Maintain the style of passing a parameter formed by the bitwise inclusive OR of flags to the initialization routines [*pthread_create*(), *pthread_mutex_init*(), *pthread_cond_init*()]. The parameter containging 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-specific 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

a) 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 (i.e., get- or set- functions), then the bitmask is merely an implementation of attributes objects as currently defined and should not be exposed to the programmer.

b) Many attributes are not Boolean or very small integral values. For example, scheduling policy may be placed in 3 b or 4 b, but priority requires 5 b or more, thereby taking up at least 8 b out of a possible 16 b on machines with 16-b 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 nonextensible), or by setting a structure field (which is non-opaque), or by get and set functions (making the bit mask 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 standard has to be done with care so as not to affect binary compatibility.

Attribute objects, even if allocated by means of dynamic allocation functions such as *malloc*(), may have their size fixed at compile time. This means, for example, a *pthread_create*() in an implementation with extensions to the *pthread_attr_t* cannot look beyond the area that the binary application assumes is valid. This suggests that implementations should maintain a size field in the attributes object, as well as possibly version information, if extensions in different directions (possibly by different vendors) are to be accommodated.

B.16.2.2 Thread Creation

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 working group 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 startup 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 will be signalled 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*() will need to wait on a condition variable to receive the order to begin “activation.” The second approach requires no such condition variable or extra synchronization. In either approach, a separate Ada task control block would need to be created when the task object is created to hold rendezvous queues, etc.

An extension of the preceding model would be to allow the state of the thread to be modified between the create and start. This would allow the thread attributes object to be eliminated. This has been rejected because

— All state in the thread attributes object has to be able to be set for the thread. This would require the definition of interfaces to modify thread attributes. There would be no reduction in the number of function calls required to set up the thread. In fact, for an application that creates all threads using identical attributes, the number of function calls required to set up the threads would be dramatically increased. Use of a thread attributes object

permits the application to make one set of attribute setting function calls. Otherwise, the set of attribute setting function calls needs to be made for each thread creation.
- Depending on the implementation archite$cture, interfaces to set thread state would require kernel calls, or for other implementation reasons would not be able to be implemented as macros, thereby increasing the cost of thread creation.
- The ability for applications to segregate threads by class would be lost.

Another suggested alternative uses a model similar to that for process creation, such as "thread fork." The fork semantics would provide more flexibility and the "create" function can be implemented simply by doing a thread fork followed immediately by a call to the desired "start routine" for the thread. This alternative has these problems:

- For many implementations, the entire stack of the calling thread would need to be duplicated, since in many architectures there is no way to determine the size of the calling frame.
- Efficiency is reduced since at least some part of the stack has to be copied, even though in most cases the thread will never need the copied context, since it will merely call the desired start routine.

B.16.2.3 Wait for Thread Termination

The *pthread_join*() function is a convenience that has proven useful in multithreaded 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 the standard 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:

1) The *pthread_join*() function, like all other non-async-cancel-safe functions, can only be called with deferred cancelability type.
2) 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 cancelled, 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.

Here is an example of thread creation and deletion.

```
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]++;
        }
        main()
        {
                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);
        }
```

B.16.2.4 Detaching a Thread

The *pthread_join*() or *pthread_detach*() functions should eventually be called for every thread that is created so that storage associated with the thread may be reclaimed.

It has been suggested that a “detach” function is not necessary—that the detach-state thread creation attribute is sufficient, since a thread need never be dynamically detached. However, need arises in at least two cases:

1) In a cancellation handler for a *pthread_join*() it is nearly essential to have a *pthread_detach*() function in order to detach the thread on which *pthread_join*() was waiting. Without it, it would be necessary to have the handler do another *pthread_join*() to attempt to detach the thread, which would both delay the cancellation processing for an unbounded period and introduce a new call to *pthread_join*(), which might itself need a cancellation handler. A dynamic detach is nearly essential in this case.
2) In order to detach the “initial thread” (as may be desirable in processes that set up server threads).

B.16.2.5 Thread Termination

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 will be executed, and the destructors for any existing thread specific data will be executed. The standard 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.

B.16.2.6 Get Thread ID

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.

B.16.2.7 Compare Thread IDs

Implementations may choose to define a thread ID as a structure. This allows additional flexibility and robustness over using an *int*. For example, a thread ID could include a sequence number that allows detection of “dangling IDs” (copies of a thread ID that has been detached). Because the C language does not support comparison on structure types, the *pthread_equal*() function is provided to compare thread IDs.

B.16.2.8 Dynamic Package Initialization

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, like this:

```
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 multithreaded 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 multithreaded 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 multithreaded 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>`.

B.16.2.9 Omitted and Rejected Functions

B.16.2.9.1 Thread Suspend/Resume

This standard does not provide a facility for suspending or resuming an individual thread. The absence of this feature does not imply that these facilities will not be available for use by debuggers, language runtimes, and other system management tools whose needs are beyond the scope of this application portability interface.

Suspend/resume is considered error-prone when generally used as a thread synchronization mechanism by applications, as there are race conditions that are difficult to avoid. In addition, there is a large amount of existing practice that does not provide these interfaces in application thread packages. Therefore, it was thought that these functions should not be made available at the application interface.

The functions “suspend self” and “resume thread” can be implemented with the interfaces defined in this standard, although perhaps not as efficiently as possible, as follows:

```
struct susp {
        struct susp     *next;
        pthread_t       who;
        pthread_cond_t  w;
} *list;
pthread_mutex_t lock;
void suspend_self(void)
{
        struct susp     s;
        pthread_mutex_lock (&lock);
        pthread_cond_init (&s.w, (pthread_condattr_t *) NULL);
        s.who = pthread_self ();
        s.next = list;
        list = &s;
        while (pthread_equal (s.who, pthread_self ()))
                pthread_cond_wait (&s.w, &lock);
        pthread_mutex_unlock (&lock);
}
void resume (pthread_t t)
{
        struct susp *p, *q;
        pthread_mutex_lock (&lock);
        for (q = (struct susp *)&list; p = q->next; q = p)
        {
                if (pthread_equal (t, p->who))
                {
                       q->next = p->next;
                       p->who = pthread_self ();
                       pthread_cond_signal (&p->w);
                       break;
                }
        }
        pthread_mutex_unlock (&lock);
}
```

B.17 Thread-Specific Data

Many applications require that a certain amount of context be maintained on a per-thread basis across procedure calls. A common example is a multithreaded library routine that allocates resources from a common pool and maintains an active resource list for each thread. The thread-specific data interface provided to meet these needs may be viewed as a two-dimensional array of values with keys serving as the row index and thread IDs as the column index (although the implementation need not work this way).

Models

Three possible thread-specific data models were considered:

1) *No explicit support*: A standard thread-specific data interface is not strictly necessary to support applications that require per-thread context. One could, for example, provide a hash function that converted a *p-thread_t* into an integer value that could then be used to index into a global array of per-thread data pointers. This hash function, in conjunction with *pthread_self*(), would be all the interface required to support a mechanism of this sort. Unfortunately, this technique is cumbersome. It can lead to duplicated code as each set of cooperating modules implements their own per-thread data management schemes.
2) *Single (void *) pointer*: Another technique would be to provide a single word of per-thread storage and a pair of functions to fetch and store the value of this word. The word could then hold a pointer to a block of per-thread memory. The allocation, partitioning, and general use of this memory would be entirely up to the application. Although this method is not as problematic as technique 1, it suffers from interoperability problems. For example, all modules using the per-thread pointer would have to agree on a common usage protocol.
3) *Key/value mechanism*: This method associates an opaque key (e.g., stored in a variable of type *pthread_key_t*) with each per-thread datum. These keys play the role of identifiers for per-thread data. This technique is the most generic and avoids the problems noted above, albeit at the cost of some complexity.

The primary advantage of the third model is its information hiding properties. Modules using this model are free to create and use their own key(s) independent of all other such usage, whereas the other models require that all modules that use thread-specific context explicitly cooperate with all other such modules. The data independence provided by the third model is worth the additional interface.

Requirements

It is important that it be possible to implement the thread-specific data interface without the use of thread private memory. To do otherwise would increase the weight of each thread, thereby limiting the range of applications for which the threads interfaces provided by this standard is appropriate.

The values that one binds to the key via *pthread_setspecific*() may, in fact, be pointers to shared storage locations available to all threads. It is only the key/value bindings that are maintained on a per-thread basis, and these can be kept in any portion of the address space that is reserved for use by the calling thread (for example, on the stack). Thus, no per-thread MMU state is required to implement the interface. On the other hand, there is nothing in the interface specification to preclude the use of a per-thread MMU state if it is available [for example, the key values returned by *pthread_key_create*() could be thread private memory addresses].

Standardization Issues

Thread-specific data is a requirement for a usable thread interface. The binding described in this section provides a portable thread-specific data mechanism for languages that do not directly support a thread-specific storage class. A binding to this standard for a language that does include such a storage class need not provide this specific interface.

If a language were to include the notion of thread-specific storage, it would be desirable (but *not* required) to provide an implementation of the pthreads thread-specific data interface based on the language feature. For example, assume that a compiler for a C-like language supports a “private” storage class that provides thread-specific storage. Something similar to the following macros might be used to effect a compatible implementation:

```
#define pthread_key_t                        private void *
#define pthread_key_create(key)              /* no-op */
#define pthread_setspecific(key, value)      (key) = (value)
#define pthread_getspecific(key)             (key)
```

NOTE — For the sake of clarity, this example ignores destructor functions. A correct implementation would have to support them.

B.17.1 Thread-Specific Data Functions

B.17.1.1 Thread-Specific Data Key Creation

Destructor Functions

Normally, the value bound to a key on behalf of a particular thread will be 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 8966 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 will 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 it 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.

B.17.1.2 Thread-Specific Data Management

Performance and ease of use of *pthread_getspecific*() will be critical for functions that rely on maintaining state in thread-specific data. Since no errors are required to be detected by it, and since the only error that could be detected is the use of an invalid key, the interface to *pthread_getspecific*() has been designed to favor speed and simplicity over error reporting.

B.17.1.3 Thread-Specific Data Key Deletion

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.

Portable 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 will not attempt to use it again. For instance, 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 zeroing a reference count.

B.17.2 Thread-Specific Data Example

The following example demonstrates a function that initializes a thread-specific data key when it is first called and associates a thread-specific object with each calling thread, initializing this object when necessary.

```
static pthread_key_t key;
static pthread_once_t key_once = PTHREAD_ONCE_INIT;
static void
make_key()
{
    (void) pthread_key_create(&key, NULL);
}
func()
```

```
{
    void *ptr;
    (void) pthread_once(&key_once, make_key);
    if ((ptr = pthread_getspecific(key)) == NULL) {
        ptr = malloc(OBJECT_SIZE);
        ...
        (void) pthread_setspecific(key, ptr);
    }
    ...
}
```

Note that the key has to be initialized before *pthread_getspecific*() or *pthread_setspecific*() can be used. The *pthread_key_create*() call could either be explicitly made in a module initialization routine, or it can be done implicitly by the first call to a module as in this example. Any attempt to use the key before it is initialized is a programming error, making the code below incorrect:

```
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);
        ...
    }
}
```

B.18 Thread Cancellation

Many existing threads packages have facilities for canceling an operation or canceling a thread. These facilities are used for implementing user requests (such as the CANCEL button in a window-based application), for implementing OR parallelism (for example, telling the other threads to stop working once one thread has found a forced mate in a parallel chess program), or for implementing the ABORT mechanism in Ada.

POSIX programs traditionally have used the signal mechanism combined with either *longjmp*() or polling to cancel operations. Many POSIX programmers have trouble using these facilities to solve their problems efficiently in a single-threaded process. With the introduction of threads, these solutions become even more difficult to use.

The main issues with implementing a cancellation facility are specifying the operation to be canceled, cleanly releasing any resources allocated to that operation, controlling when the target notices that it has been canceled, and defining the interaction between asynchronous signals and cancellation.

Specifying the Operation to Cancel

Consider a thread that calls through five distinct levels of program abstraction and then, inside the lowest level abstraction, calls a function that suspends the thread. (An abstraction boundary is a layer at which the client of the abstraction sees only the service being provided and can remain ignorant of the implementation. Abstractions are often layered, each level of abstraction being a client of the lower level abstraction and implementing a higher level abstraction.) Depending on the semantics of each abstraction, one could imagine wanting to cancel only the call that causes suspension, only the bottom two levels, or the operation being done by the entire thread. Canceling operations at a finer grain than the entire thread is difficult because threads are active and they may be run in parallel on a multiprocessor. By the time one thread can make a request to cancel an operation, the thread performing the operation may have completed that operation and gone on to start another operation whose cancellation is not desired. Thread

IDs are not reused until the thread has exited, and either it was created with the detachstate attribute set to PTHREAD_CREATE_DETACHED or the *pthread_join*() or *pthread_detach*() function has been called for that thread. Consequently, a thread cancellation will never be misdirected when the thread terminates. For these reasons, the canceling of operations is done at the granularity of the thread. Threads are designed to be inexpensive enough so that a separate thread may be created to perform each separately cancelable operation, for example, each possibly long running user request.

For cancellation to be used in existing code, cancellation scopes and handlers will have to be established for code that needs to release resources upon cancellation, so that it follows the programming discipline described in the text.

A Special Signal Versus a Special Interface

Two different mechanisms were considered for providing the cancellation interfaces. The first was to provide an interface to direct signals at a thread and then to define a special signal that had the required semantics. The other alternative was to use a special interface that delivered the correct semantics to the target thread.

The solution using signals produced a number of problems. It required the implementation to provide cancellation in terms of signals whereas a perfectly valid (and possibly more efficient) implementation could have both layered on a low-level set of primitives. There were so many exceptions to the special signal (it cannot be used with kill, no POSIX.1 interfaces can be used with it) that it was clearly not a valid signal. Its semantics on delivery were also completely different from any existing POSIX.1 signal. As such, a special interface that did not mandate the implementation and did not confuse the semantics of signals and cancellation was felt to be the better solution.

Races Between Cancellation and Resuming Execution

Due to the nature of cancellation, there is generally no synchronization between the thread requesting the cancellation of a blocked thread and events that may cause that thread to resume execution. For this reason, and because excess serialization hurts performance, when both an event that a thread is waiting for has occurred and a cancellation request has been made and cancellation is enabled, the standard explicitly allows the implementation to choose between returning from the blocking call or acting on the cancellation request.

Interaction of Cancellation With Asynchronous Signals

A typical use of cancellation is to acquire a lock on some resource and to establish a cancellation cleanup handler for releasing the resource when and if the thread is canceled.

A correct and complete implementation of cancellation in the presence of asynchonous signals requires considerable care. An implementation has to push a cancellation cleanup handler on the cancellation cleanup stack while maintaining the integrity of the stack data structure. If an asynchronously generated signal is posted to the thread during a stack operation, the signal handler cannot manipulate the cancellation cleanup stack. As a consequence, asynchronous signal handlers may not cancel threads or otherwise manipulate the cancellation state of a thread. Threads may, of course, be canceled by another thread that used a *sigwait* function to wait synchronously for an asynchronous signal.

In order for cancellation to function correctly, it is required that asynchronous signal handlers not change the cancellation state. This requires that some elements of existing practice, such as using *longjmp*() to exit from an asynchronous signal handler implicitly, be prohibited in cases where the integrity of the cancellation state of the interrupt thread cannot be ensured.

B.18.1 Thread Cancellation Overview

B.18.1.1 Cancelability States

The three possible cancelability states (disabled, deferred, and asynchronous) are encoded into two separate bits [(disable, enable) and (deferred, asynchronous)] to allow them to be changed and restored independently. For instance, short code sequences that will not block sometimes disable cancelability on entry and restore the previous state upon exit. Likewise, long or unbounded code sequences containing no convenient explicit cancellation points will sometimes set the cancelability type to asynchronous on entry and restore the previous value upon exit.

B.18.1.2 Cancellation Points

Cancellation points are points inside of certain functions where a thread has to act on any pending cancellation request when cancelability is enabled, if the function would block. As with checking for signals, operations need only check for pending cancellation requests when the operation is about to block indefinitely.

The idea was considered of allowing implementations to define whether blocking calls such as *read*() should be cancellation points. It was decided that it would adversely affect the design of portable applications if blocking calls were not cancellation points because threads could be left blocked in an uncancelable state.

There is one important blocking routine that is specifically not made a cancellation point: *pthread_mutex_lock*(). If *pthread_mutex_lock*() were a cancellation point, every routine that called it would also become a cancellation point (that is, any routine that touched shared state would automatically become a cancellation point). For example, *malloc*(), *free*(), and *rand*() would become cancellation points under this scheme. Having too many cancellation points makes programming very difficult, leading to either much disabling and restoring of cancelability or much difficulty in trying to arrange for reliable cleanup at every possible place.

Since *pthread_mutex_lock*() is not a cancellation point, threads could result in being blocked uninterruptibly for long periods of time if mutexes were used as a general synchronization mechanism. As this is normally not acceptable, mutexes should only be used to protect resources that are held for small fixed lengths of time where not being able to be cancelled will not be a problem. Resources that need to be held exclusively for long periods of time should be protected with condition variables.

Every library routine should specify whether or not it includes any cancellation points. Typically, only those routines that may block or compute indefinitely need to include cancellation points.

Correctly coded routines only reach cancellation points after having set up a cancellation cleanup handler to restore invariants if the thread is canceled at that point. Being cancelable only at specified cancellation points allows programmers to keep track of actions needed in a cancellation cleanup handler more easily. A thread should only be made asynchronously cancelable when it is not in the process of acquiring or releasing resources or otherwise in a state from which it would be difficult or impossible to recover.

B.18.1.3 Thread Cancellation Cleanup Handlers

The cancellation cleanup handlers provide a portable mechanism, easy to implement, for releasing resources and restoring invariants. They are easier to use than signal handlers because they provide a stack of cancellation cleanup handlers rather than a single handler and because they have an argument that can be used to pass context information to the handler.

The alternative to providing these simple cancellation cleanup handlers (whose only use is for cleaning up when a thread is canceled) is to define a general exception package that could be used for handling and cleaning up after hardware traps and software detected errors. This was too far removed from the charter of providing threads to handle asynchrony. However, it is an explicit goal of this standard to be compatible with existing exception facilities and languages having exceptions.

The interaction of this facility and other procedure-based or language-level exception facilities is unspecified in this version of this standard. However, it is intended that it be possible for an implementation to define the relationship between these cancellation cleanup handlers and Ada, C++, or other language-level exception handling facilities.

It was suggested that the cancellation cleanup handlers should also be called when the process exits or calls the exec function. This was rejected partly due to the performance problem caused by having to call the cancellation cleanup handlers of every thread before the operation could continue. The other reason was that the only state expected to be cleaned up by the cancellation cleanup handlers would be the intraprocess state. Any handlers that are to clean up the interprocess state would be registered with *atexit*(). There is the orthogonal problem that the exec functions do not honor the *atexit*() handlers, but resolving this is beyond the scope of this standard.

B.18.1.4 Async-Cancel Safety

A function is said to be *async-cancel safe* if it is written in such a way that entering the function with asynchronous cancelability enabled will not cause any invariants to be violated, even if a cancellation request is delivered at any arbitrary instruction. Functions that are async-cancel safe are often written in such a way that they need to acquire no resources for their operation and the visible variables that they may write are strictly limited.

Any routine that gets a resource as a side effect cannot be made async-cancel safe [for example, *malloc*()]. If such a routine were called with asynchronous cancelability enabled, it might acquire the resource successfully, but as it was returning to the client, it could act on a cancellation request. In such a case, the application would have no way of knowing whether the resource was acquired or not.

Indeed, because many interesting routines cannot be made async-cancel safe, most library routines in general are not async-cancel safe. Every library routine should specify whether or not it is async-cancel safe so that programmers know which routines can be called from code that is asynchronously cancelable.

B.18.2 Thread Cancellation Functions

B.18.2.1 Canceling Execution of a Thread

Two alternative interfaces were considered to sending the cancellation notification to a thread. One would be to define a new SIGCANCEL signal that had the cancellation semantics when delivered; the other was to define the new *pthread_cancel*() function, which would trigger the cancellation semantics.

The advantage of a new signal was that so much of the delivery criteria were identical to that used when trying to deliver a signal that making cancellation notification a signal was seen as consistent. Indeed, many implementations will implement cancellation using a special signal. On the other hand, there would be no signal functions that could be used with this signal except *pthread_kill*(), and the behavior of the delivered cancellation signal would be unlike any previously existing defined signal.

The benefits of a special interface include the recognition that this signal would be defined because of the similar delivery criteria and that this is the only common behavior between a cancellation request and a signal. In addition, the cancellation delivery mechanism does not have to be implemented as a signal. There are also strong, if not stronger, parallels with language exception mechanisms than with signals that are potentially obscured if the delivery mechanism is visibly closer to signals.

In the end, it was considered that as there were so many exceptions to the use of the new signal with existing signals interfaces that it would be misleading. A special interface has resolved this problem. This interface was carefully defined so that an implementation wishing to provide the cancellation interfaces on top of signals could do so. The special interface also means that implementations are not obliged to implement cancellation with signals.

B.18.2.2 Setting Cancelability State

The *pthread_setcancelstate*() and *pthread_setcanceltype*() functions are used to 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 will be 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.

B.18.2.3 Establishing Cancellation Handlers

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. The restriction that they have to appear in the same lexical scope allows for efficient macro or compiler implementations and efficient storage management. A sample implementation of these routines as macros might look like

```
#define pthread_cleanup_push(rtn,arg) { \
        struct _pthread_handler_rec __cleanup_handler, **__head; \
        __cleanup_handler.rtn = rtn; \
        __cleanup_handler.arg = arg; \
        (void) pthread_getspecific(_pthread_handler_key, &__head); \
        __cleanup_handler.next = *__head; \
        *__head = &__cleanup_handler;
#define pthread_cleanup_pop(ex) \
        *__head = __cleanup_handler.next; \
        if (ex) (*__cleanup_handler.rtn) (__cleanup_handler.arg); \
}
```

A more ambitious implementation of these routines might do even better by allowing the compiler to note that the cancellation cleanup handler is a constant and can be expanded inline.

This standard currently leaves unspecified the effect of calling *longjmp*() from a signal handler executing in a POSIX.1 function. If an implementation wants to allow this and give the programmer reasonable behavior, the *longjmp*() function has to call all cancellation cleanup handlers that have been pushed but not popped since the time *setjmp*() was called.

Consider a multithreaded function called by a thread that uses signals. If a signal were delivered to a signal handler during the operation of *qsort*() and that handler were to call *longjmp*() (which, in turn, did *not* call the cancellation cleanup handlers) the helper threads created by the *qsort*() function would not be canceled. Instead, they would continue to execute and write into the argument array even though the array might have been popped off of the stack.

Note that the specified cleanup handling mechanism is especially tied to the C language and, while the requirement for a uniform mechanism for expressing clean up is language independent, the mechanism used in other languages may be quite different. In addition, this mechanism is really only necessary due to the lack of a real exception mechanism in the C language, which would be the ideal solution.

There is no notion of a cancellation cleanup-safe function. If an application has no cancellation points in its signal handlers, blocks any signal whose handler may have cancellation points while calling async-unsafe functions, or disables cancellation while calling async-unsafe functions, all functions may be safely called from cancellation cleanup routines.

B.18.2.3.1 Examples

The following is an example using thread primitives to implement a cancelable, writers-priority readers/writers lock:

```
typedef struct {
        pthread_mutex_t lock;
        pthread_cond_t rcond,
                       wcond;
        int lock_count; /* < 0 .. held by writer             */
                        /* > 0 .. held by lock_count readers */
                        /* = 0 .. held by nobody             */
        int waiting_writers; /* count of waiting writers     */
} rwlock;
void
waiting_reader_cleanup(void *arg)
{
        rwlock *l;
        l = (rwlock *) arg;
        pthread_mutex_unlock(&l->lock);
}
void
lock_for_read(rwlock *l)
{
        pthread_mutex_lock(&l->lock);
        pthread_cleanup_push{waiting_reader_cleanup, l);
        while ((l->lock_count < 0) && (l->waiting_writers ! = 0))
                pthread_cond_wait(&l->rcond, &l->lock);
        l->lock count++;
        /*
         * Note the pthread_cleanup_pop will execute
         * waiting_reader_cleanup
         */
        pthread_cleanup_pop(1);
}
void
release_read_lock(rwlock *l)
{
        pthread_mutex_lock(&l->lock);
        if (--l->lock_count == 0)
                pthread_cond_signal(&l->wcond);
        pthread_mutex_unlock(l);
```

```
}
void
waiting_writer_cleanup(void *arg)
{
        rwlock *l;
        l = (rwlock *) arg;
        if ((--l->waiting_writers == 0) && (l->lock_count >= 0)) {
                /*
                 * This only happens if we have been canceled
                 */
                pthread_cond_broadcast(&l->wcond);
}
        pthread_mutex_unlock(&l->lock);
}
void
lock_for_write(rwlock *l)
{
        pthread_mutex_lock(&l->lock);
        l->waiting_writers++;
        pthread_cleanup_push(waiting_writer_cleanup, l);
        while (l->lock_count != 0)
                pthread_cond_wait(&l->wcond, &l->lock);
        l->lock_count = –1;
        /*
         * Note the pthread_cleanup_pop will execute
         * 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);
        }
```

B.18.3 Language-Independent Cancellation Functionality

The language-independent cancellation functionality is intended to provide each language binding with the capability for active computations to be canceled in a controlled fashion by other computations. Cleanup code has to be associated with scopes so that resources held by scopes can be freed and scope invariants can be maintained when cancellation occurs. Cancellation has to occur only at well-defined points under control of the scopes to be canceled in order for the complexity of scope cleanup to remain feasible.

It is intended that bindings be able to use language exception facilities as part of the implementation of thread cancellation. In particular, it would be desirable to have thread cancellation, cancellation scopes, and their associated cleanup code map into exception raise, exception scopes, and exception handlers in languages providing them.

Some bindings may choose to allow active cancellations to be caught and declared completed by cleanup code, allowing execution of the cancelled thread to resume normally from that point on. This is a natural operation to allow for bindings that map cancellations into exceptions and cleanup code into exception handlers. Note that this operation may require that the stack be unwound during cancellation. By contrast, if all cleanup code is run and the thread is terminated, the stack may not need to be unwound.

Annex C Header Contents Samples

(Informative)

The material in this informative annex serves as an index to which symbols should appear in which headers in a system that conforms to POSIX.1 with C Standard Language-Dependent System support.

This is only an index, and any conflicts with the actual body of any relevant standard shall be resolved in favor of that standard. The actual body of the declaration was omitted in part because this is an index and in part to avoid any possible conflict with the standards.

Where it is known that a symbol or header is not required for Common Usage C Language-Dependent System support, the name is followed by an asterisk (*). Omission of an asterisk does not imply that the symbol is required for Common Usage C. For Common-Usage C, although the location of symbols is typical, it is not to be taken as a requirement: POSIX.1 is quite explicit that there is no requirement except that differences from the C Standard be documented. {2}

Generally, where it is stated that functions are defined in a header, macros are permitted as acceptable alternatives by both standards. See the bodies of the standards for details.

`<aio.h>`

The header defines the macros

AIO_ALLDONE	LIO_NOP	LIO_WAIT
AIO_CANCELED	LIO_NOWAIT	LIO_WRITE
AIO_NOTCANCELED	LIO_READ	

and the structure

aiocb

with structure elements

aio_buf	*aio_lio_opcode*	*aio_offset*	*aio_sigevent*
aio_fildes	*aio_nbytes*	*aio_reqprio*	

and the functions

aio_cancel()	*aio_fsync*()	*aio_return*()	*aio_write*()
aio_error()	*aio_read*()	*aio_suspend*()	*lio_listio*()

`<assert.h>`

The header defines the macro

assert()

and makes reference to the macro

NDEBUG

`<ctype.h>`

The header declares the functions

isalnum()	*isdigit*()	*islower*()	*ispunct*()	*isupper*()	*tolower*()
isalpha()	*isgraph*()	*isprint*()	*isspace*()	*isxdigit*()	*toupper*()
iscntrl()					

`<dirent.h>`

The header defines the typedef

DIR

and declares the structure

dirent

with structure element member

d_name

and declares functions

closedir() *opendir*() *readdir*() *readdir_r*() *rewinddir*()

`<errno.h>`

The header defines the macros

E2BIG	EFAULT	ENFILE	ENOTTY
EACCES	EFBIG	ENODEV	ENXIO
EAGAIN	EINPROGRESS	ENOENT	EPERM
EBADF	EINTR	ENOEXEC	EPIPE
EBADMSG	EINVAL	ENOLCK	ERANGE
EBUSY	EIO	ENOMEM	EROFS
ECANCELED	EISDIR	ENOSPC	ESPIPE
ECHILD	EMFILE	ENOSYS	ESRCH
EDEADLK	EMLINK	ENOTDIR	ETIMEDOUT
EDOM	EMSGSIZE	ENOTEMPTY	EXDEV
EEXIST	ENAMETOOLONG	ENOTSUP	

and defines the symbol

errno

`<fcntl.h>`

The header defines the macros

FD_CLOEXEC	F_SETFL	O_APPEND	O_RDONLY
F_DUPFD	F_SETLK	O_CREAT	O_RDWR
F_GETFD	F_SETLKW	O_DSYNC	O_RSYNC
F_GETFL	F_UNLCK	O_EXCL	O_SYNC

F_GETLK	F_WRLCK	O_NOCTTY	O_TRUNC
F_RDLCK	O_ACCMODE	O_NONBLOCK	O_WRONLY
F_SETFD			

and declares the structure

flock

with structure elements

l_len *l_pid* *l_start* *l_type* *l_whence*

and the functions

creat() *fcntl*() *open*()

and may contain the macros

SEEK_CUR	S_IRUSR	S_ISCHR	S_ISREG	S_IWUSR
SEEK_END	S_IRWXG	S_ISDIR	S_ISUID	S_IXGRP
SEEK_SET	S_IRWXO	S_ISFIFO	S_IWGRP	S_IXOTH
S_IRGRP	S_IRWXU	S_ISGID	S_IWOTH	S_IXUSR
S_IROTH	S_ISBLK			

`<float.h>`

The header defines the macros

DBL_DIG*	FLT_EPSILON*	LDBL_DIG*
DBL_EPSILON*	FLT_MANT_DIG*	LDBL_EPSILON*
DBL_MANT_DIG*	FLT_MAX*	LDBL_MANT_DIG*
DBL_MAX*	FLT_MAX_10_EXP*	LDBL_MAX*
DBL_MAX_10_EXP*	FLT_MAX_EXP*	LDBL_MAX_10_EXP*
DBL_MAX_EXP*	FLT_MIN*	LDBL_MAX_EXP*
DBL_MIN*	FLT_MIN_10_EXP*	LDBL_MIN*
DBL_MIN_10_EXP*	FLT_MIN_EXP*	LDBL_MIN_10_EXP*
DBL_MIN_EXP*	FLT_RADIX*	LDBL_MIN_EXP*
FLT_DIG*	FLT_ROUNDS*	

`<grp.h>`

The header declares the structure

group

with structure elements

gr_gid *gr_mem* *gr_name*

and the functions

getgrgid() *getgrnam*() |
getgrgid_r() *getgrnam_r*()

`<limits.h>`

The header defines the macros

_POSIX_AIO_LISTIO_MAX	_POSIX_OPEN_MAX
_POSIX_AIO_MAX	_POSIX_PATH_MAX
_POSIX_ARG_MAX	_POSIX_PIPE_BUF
_POSIX_CHILD_MAX	_POSIX_RTSIG_MAX
_POSIX_CLOCKRES_MIN	_POSIX_SEM_NSEMS_MAX
_POSIX_DELAYTIMER_MAX	_POSIX_SEM_VALUE_MAX
_POSIX_LINK_MAX	_POSIX_SIGQUEUE_MAX
_POSIX_LOGIN_NAME_MAX	_POSIX_SSIZE_MAX
_POSIX_MAX_CANON	_POSIX_STREAM_MAX
_POSIX_MAX_INPUT	_POSIX_THREAD_KEYS_MAX
_POSIX_MQ_OPEN_MAX	_POSIX_THREAD_THREADS_MAX
_POSIX_MQ_PRIO_MAX	_POSIX_TIMER_MAX
_POSIX_NAME_MAX	_POSIX_TTY_NAME_MAX
_POSIX_NGROUPS_MAX	_POSIX_TZNAME_MAX

_POSIX_THREAD_DESTRUCTOR_ITERATIONS

and defines the macros

AIO_LISTIO_MAX □	MAX_CANON•
AIO_MAX □	MAX_INPUT•
ARG_MAX □	MB_LEN_MAX
CHAR_BIT	MQ_OPEN_MAX □
CHAR_MAX	MQ_PRIO_MAX•
CHAR_MIN	NAME_MAX•
CHILD_MAX □	NGROUPS_MAX
DELAYTIMER_MAX•	OPEN_MAX □
INT_MAX	PAGESIZE
INT_MIN	PATH_MAX•
LINK_MAX•	PIPE_BUF•
LOGIN_NAME_MAX □	RTSIG_MAX □
LONG_MAX	SCHAR_MAX
LONG_MIN	SCHAR_MIN

PTHREAD_DESTRUCTOR_ITERATIONS □
PTHREAD_KEYS_MAX □
PTHREAD_STACK_MIN □
PTHREAD_THREADS_MAX □

The macros marked with □ shall be omitted from `<limits.h>` on specific implementations where the corresponding value is greater than or equal to the stated minimum, but is indeterminate. The macros marked with • shall be omitted from `<limits.h>` on specific implementations where the corresponding value is greater than or equal to the stated minimum, but where the value can vary depending on the file to which it is applied.

`<locale.h>`

The header defines the macros

LC_ALL*	LC_CTYPE*	LC_NUMERIC*	NULL*
LC_COLLATE*	LC_MONETARY*	LC_TIME*	

and declares the structure

*lconv**

with structure elements

*currency_symbol**	*mon_decimal_point**	*negative_sign**
*decimal_point**	*mon_grouping**	*p_cs_precedes**
*frac_digits**	*mon_thousands_sep**	*p_sep_by_space**
*grouping**	*n_cs_precedes**	*p_sign_posn**
*int_curr_symbol**	*n_sep_by_space**	*positive_sign**
*int_frac_digits**	*n_sign_posn**	*thousands_sep**

and the functions

localeconv()* *setlocale*()*

`<math.h>`

The header defines the macro

HUGE_VAL

and declares the functions

acos()	*ceil*()	*exp*()	*fmod*()	*log10*()	*pow*()	*sqrt*()
asin()	*cos*()	*fabs*()	*frexp*()	*log*()	*sin*()	*tan*()
atan2()	*cosh*()	*floor*()	*ldexp*()	*modf*()	*sinh*()	*tanh*()
atan()						

`<mqueue.h>`

The header defines the type

mqd_t

and the structure *sigevent* as defined in `<signal.h>`

and the structure

mq_attr

with structure elements

mq_curmsgs *mq_flags* *mq_maxmsg* *mq_msgsize*

and the functions

mq_close()	*mq_notify*()	*mq_receive*()	*mq_setattr*()
mq_getattr()	*mq_open*()	*mq_send*()	*mq_unlink*()

`<pthread.h>`

The header defines the functions

pthread_atfork()	*pthread_detach*()
pthread_attr_destroy()	*pthread_equal*()

pthread_attr_getdetachstate()	*pthread_exit*()
pthread_attr_getinheritsched()	*pthread_getspecific*()
pthread_attr_getschedparam()	*pthread_join*()
pthread_attr_getschedpolicy()	*pthread_key_create*()
pthread_attr_getscope()	*pthread_key_delete*()
pthread_attr_getstackaddr()	*pthread_kill*()
pthread_attr_getstacksize()	*pthread_mutex_destroy*()
pthread_attr_init()	*pthread_mutex_getprioceiling*()
pthread_attr_setdetachstate()	*pthread_mutex_init*()
pthread_attr_setinheritsched()	*pthread_mutex_lock*()
pthread_attr_setschedparam()	*pthread_mutex_setprioceiling*()
pthread_attr_setschedpolicy()	*pthread_mutex_trylock*()
pthread_attr_setscope()	*pthread_mutex_unlock*()
pthread_attr_setstackaddr()	*pthread_mutexattr_destroy*()
pthread_attr_setstacksize()	*pthread_mutexattr_getprioceiling*()
pthread_cleanup_pop()	*pthread_mutexattr_getprotocol*()
pthread_cleanup_push()	*pthread_mutexattr_getpshared*()
pthread_cond_broadcast()	*pthread_mutexattr_init*()
pthread_cond_destroy()	*pthread_mutexattr_setprioceiling*()
pthread_cond_init()	*pthread_mutexattr_setprotocol*()
pthread_cond_signal()	*pthread_mutexattr_setpshared*()
pthread_cond_timedwait()	*pthread_once*()
pthread_cond_wait()	*pthread_self*()
pthread_condattr_destroy()	*pthread_setcancelstate*()
pthread_condattr_getpshared()	*pthread_setcanceltype*()
pthread_condattr_init()	*pthread_setspecific*()
pthread_condattr_setpshared()	*pthread_sigmask*()
pthread_create()	*pthread_testcancel*()

and defines the macros

PTHREAD_CANCELED	PTHREAD_MUTEX_INITIALIZER
PTHREAD_CANCEL_ASYNCHRONOUS	PTHREAD_ONCE_INIT
PTHREAD_CANCEL_DEFERRED	PTHREAD_PRIO_INHERIT
PTHREAD_CANCEL_DISABLE	PTHREAD_PRIO_NONE
PTHREAD_CANCEL_ENABLE	PTHREAD_PRIO_PROTECT
PTHREAD_COND_INITIALIZER	PTHREAD_PROCESS_PRIVATE
PTHREAD_CREATE_DETACHED	PTHREAD_PROCESS_SHARED
PTHREAD_CREATE_JOINABLE	PTHREAD_SCOPE_PROCESS
PTHREAD_EXPLICIT_SCHED	PTHREAD_SCOPE_SYSTEM
PTHREAD_INHERIT_SCHED	

`<pwd.h>`

The header defines the structure

passwd

with structure elements

pw_dir *pw_gid* *pw_name* *pw_shell* *pw_uid*

and declares the functions

getpwnam() *getpwuid*()

getpwnam_r() *getpwuid_r*()

<sched.h>

The header defines the macros

SCHED_FIFO SCHED_OTHER SCHED_RR

and the structure

sched_param

with structure elements

sched_priority

and the functions

sched_get_priority_max()	*sched_getparam*()	*sched_setscheduler*()
sched_get_priority_min()	*sched_getscheduler*()	*sched_yield*()
sched_get_rr_interval()	*sched_setparam*()	

and the symbols defined by `<time.h>`.

<semaphore.h>

The header defines the type

sem_t

and declares the functions

sero_close()	*sem_init*()	*sem_post*()	*sem_unlink*()
sem_destroy()	*sem_open*()	*sem_trywait*()	*sem_wait*()
sem_getvalue()			

<setjmp.h>

The header defines the types

jmp_buf *sigjmp_buf*

and declares the functions

longjmp() *setjmp*() *siglongjmp*() *sigsetjmp*()

Note that the C Standard {2} and this part of ISO/IEC 9945 both permit these functions to be defined solely as macros.

<signal.h>

The header defines the macros

SA_NOCLDSTOP	SIGFPE	SIGSTOP	SIG_ERR*
SA_SIGINFO	SIGHUP	SIGTERM	SIG_IGN
SIGABRT	SIGILL	SIGTSTP	SIG_SETMASK

SIGALRM	SIGINT	SIGTTIN	SIG_UNBLOCK
SIGBUS	SIGKILL	SIGTTOU	SI_ASYNCIO
SIGCHLD	SIGPIPE	SIGUSR1	SI_MESGQ
SIGCONT	SIGQUIT	SIGUSR2	SI_QUEUE
SIGEV_NONE	SIGRTMAX	SIG_BLOCK	SI_TIMER
SIGEV_SIGNAL	SIGRTMIN	SIG_DFL	SI_USER
SIGEV_THREAD	SIGSEGV		

and the types

*sig_atomic_t** *sigset_t*

and declares the structure

sigaction

with structure elements

sa_flags *sa_handler* *sa_mask*
sa_sigaction

and defines the type

siginfo_t

with the members

si_code *si_signo* *si_value*

and declares the structure

sigevent

with the members

sigev_notify *sigev_notify_function* *sigev_notify_attributes*
sigev_signo *sigev_value*

and the union

sigval

with the members

sival_int *sival_ptr*

and the functions

kill()	*sigdelset*()	*signal*()	*sigsuspend*()
raise()	*sigemptyset*()	*sigpending*()	*sigtimedwait*()
sigaction()	*sigfillset*()	*sigprocmask*()	*sigwait*()
sigaddset()	*sigismember*()	*sigqueue*()	*sigwaitinfo*()

<stdarg.h>

The header defines the macros

*va_arg**	*va_end**	*va_list**	*va_start**

<stddef.h>

The header defines the macros

NULL*	*offsetof**

and the types

*ptrdiff_t**	*size_t**	*wchar_t**

<stdio.h>

The header defines the macros

BUFSIZ	L_tmpnam*	STREAM_MAX	*stdout*
EOF	NULL	TMP_MAX	_IOFBF*
FILENAME_MAX*	SEEK_CUR	*stderr*	_IOLBF*
L_ctermid	SEEK_END	*stdin*	_IONBF*
L_cuserid	SEEK_SET		

NOTE — The L_cuserid symbol is permitted in this header, but need not be supplied. See 2.7.2.

and the types

*fpos_t**	*size_tw*

and declares the type

FILE

and the functions

clearerr()	*fscanf*()	*putc_unlocked*()
fclose()	*fseek*()	*puts*()
fdopen()	*fsetpos*()	*remove*()
feof()	*ftell*()	*rename*()
ferror()	*ftrylockfile*()	*rewind*()
fflush()	*funlockfile*()	*scanf*()
fgetc()	*fwrite*()	*setbuf*()
fgetpos()	*getc*()	*setvbuf*()
fgets()	*getchar*()	*sprintf*()
fileno()	*getchar_unlocked*()	*sscanf*()
flockfile()	*getc_unlocked*()	*tmpfile*()
fopen()	*gets*()	*tmpnam*()
fprintf()	*perror*()	*ungetc*()
fputc()	*printf*()	*vfprintf*()
fputs()	*putc*()	*vprintf*()
fread()	*putchar*()	*vsprintf*()

freopen()	*putchar_unlocked*()

`<stdlib.h>`

The header defines the macros

EXIT_FAILURE	MB_CUR_MAX*	RAND_MAXM
EXIT_SUCCESS	NULL	

and the types

*div_t**	*ldiv_t**	*size_t**	*wchar_t**

and declares the functions

abort()	*bsearch*()	*labs*()*	*qsort*()	*strtol*()*
abs()	*calloc*()	*ldiv*()*	*rand*()	*strtoul*()*
atexit()*	*div*()*	*malloc*()	*rand_r*()	*system*()*
atof()	*exit*()	*mblen*()*	*realloc*()	*wcstombs*()*
atoi()	*free*()	*mbstowcs*()*	*srand*()	*wctomb*()*
atol()	*getenv*()	*mbtowc*()*	*strtod*()*	

`<string.h>`

The header defines the macro

NULL

and the type

size_t

and declares the functions

memchr()*	*strcat*()	*strcspn*()	*strncpy*()	*strstr*()
memcmp()*	*strchr*()	*strerror*()*	*strpbrk*()	*strtok*()
memcpy()*	*strcmp*()	*strlen*()	*strrchr*()	*strtok_r*()
memmove()*	*strcoll*()*	*strncat*()	*strspn*()	*strxfrm*()*
memset()*	*strcpy*()	*strncmp*()		

`<sys/mman.h>`

The header defines the macros

MAP_FAILED	MCL_CURRENT	MS_INVALIDATE	PROT_NONE
MAP_FIXED	MCL_FUTURE	MS_SYNC	PROT_READ
MAP_PRIVATE	MS_ASYNC	PROT_EXEC	PROT_WRITE
MAP_SHARED			

and declares the functions

mlock()	*mprotect*()	*munlockall*()	*shm_open*()
mlockall()	*msync*()	*munmap*()	*shm_unlink*
mmap()	*munlock*()		

`<sys/stat.h>`

The header defines the macros

S_IRGRP	S_ISBLK	S_ISUID	S_IXOTH
S_IROTH	S_ISCHR	S_IWGRP	S_IXUSR
S_IRUSR	S_ISDIR	S_IWOTH	S_TYPEISMQ
S_IRWXG	S_ISFIFO	S_IWUSR	S_TYPEISSEM
S_IRWXO	S_ISGID	S_IXGRP	S_TYPEISSHM
S_IRWXU	S_ISREG		

and declares the structure

stat

with structure elements

st_atime	st_dev	st_ino	st_mtime	st_size
st_ctime	st_gid	st_mode	st_nlink	st_uid

and the functions

chmod()	*fstat*()	*mkfifo*()	*umask*()
fchmod()	*mkdir*()	*stat*()	

`<sys/times.h>`

The header defines the type

clock_t

and declares the structure

tms

with structure elements

tms_cstime *tms_cutime* *tms_stime* *tms_utime*

and the function

times()

`<sys/types.h>`

The header defines the types

dev_t	*pid_t*	*pthread_mutex_t*
gid_t	*pthread_attr_t*	*pthread_once_t*
ino_t	*pthread_condattr_t*	*pthread_t*
mode_t	*pthread_cond_t*	*size_t*
nlink_t	*pthread_key_t*	*ssize_t*
off_t	*pthread_mutexattr_t*	*uid_t*

`<sys/utsname.h>`

The header declares the structure

utsname
utsname

with structure elements

machine *nodename* *release* *sysname* *version*

and the function

uname()

`<sys/wait.h>`

The header defines the macros

WEXITSTATUS	WIFSIGNALED	WNOHANG	WTERMSIG
WIFEXITED	WIFSTOPPED	WSTOPSIG	WUNTRACED

and declares the functions

wait() *waitpid*()

`<termios.h>`

The header defines the macros

B0	B75	ECHONL	NCCS	TCSAFLUSH
B110	B9600	HUPCL	NOFLSH	TCSANOW
B1200	BRKINT	ICANON	OPOST	TOSTOP
B134	CLOCAL	ICRNL	PARENB	VEOF
B150	CREAD	IEXTEN	PARMRK	VEOL
B1800	CS5	IGNBRK	PARODD	VERASE
B19200	CS6	IGNCR	TCIFLUSH	VINTR
B200	CS7	IGNPAR	TCIOFF	VKILL
B2400	CS8	INLCR	TCIOFLUSH	VMIN
B300	CSIZE	INPCK	TCION	VQUIT
B38400	CSTOPB	ISIG	TCOFLUSH	VSTART
B4800	ECHO	ISTRIP	TCOOFF	VSTOP
B50	ECHOE	IXOFF	TCOON	VSUSP
B600	ECHOK	IXON	TCSADRAIN	VTIME

and the types

cc_t *speed_t* *tcflag_t*

and declares the structure

termios

with structure elements

c_cc *c_cflag* *c_iflag* *c_lflag* *c_oflag* and the functions

cfgetispeed() *cfsetospeed*() *tcflush*() *tcsendbreak*()
cfgetospeed() *tcdrain*() *tcgetattr*() *tcsetattr*()
cfsetispeed() *tcflow*()

`<time.h>`

The header defines the macros

CLK_TCK CLOCK_REALTIME TIMER_ABSTIME
CLOCKS_PER_SEC NULL

the types

clockid_t *size_t* *time_t*
`clock_t` *`timer_t`*

and declares the structure

tm

with structure elements

tm_hour *tm_mday* *tm_mon* *tm_wday* *tm_year*
tm_isdst *tm_min* *tm_sec* *tm_yday*

and declares the structure

timespec

with the members

tv_nsec *tv_sec*

and the structure

itimerspec

with the members

it_interval *it_value*

and the functions

asctime() *difftime*() *time*()
asctime_r() *gmtime*() *timer_create*()
clock() *gmtime_r*() *timer_delete*()
clock_getres() *localtime*() *timer_getoverrun*()
clock_gettime() *localtime_r*() *timer_gettime*()
clock_settime() *mktime*() *timer_settime*()
ctime() *nanosleep*() *tzset*()
ctime_r() *strftime*()

and declares the external variable

tzname

`<unistd.h>`

The header defines the macros

F_OK	SEEK_END	STDOUT_FILENO
NULL	SEEK_SET	W_OK
R_OK	STDERR_FILENO	X_OK
SEEK_CUR	STDIN_FILENO	

and defines the macros

_POSIX_ASYNCHRONOUS_IO	_POSIX_SAVED_IDS
_POSIX_ASYNC_IO	_POSIX_SEMAPHORES
_POSIX_CHOWN_RESTRICTED	_POSIX_SHARED_MEMORY_OBJECTS
_POSIX_FSYNC	_POSIX_SYNCHRONIZED_IO
_POSIX_JOB_CONTROL	_POSIX_SYNC_IO
_POSIX_MAPPED_FILES	_POSIX_THREADS
_POSIX_MEMLOCK	_POSIX_THREAD_ATTR_STACKADDR
_POSIX_MEMLOCK_RANGE	_POSIX_THREAD_ATTR_STACKSIZE
_POSIX_MEMORY_PROTECTION	_POSIX_THREAD_PRIO_INHERIT
_POSIX_MESSAGE_PASSING	_POSIX_THREAD_PRIO_PROTECT
_POSIX_NO_TRUNC	_POSIX_THREAD_PROCESS_SHARED
_POSIX_PRIORITIZED_IO	_POSIX_THREAD_SAFE_FUNCTIONS
_POSIX_PRIORITY_SCHEDULING	_POSIX_TIMERS
_POSIX_PRIO_IO	_POSIX_VDISABLE
_POSIX_REALTIME_SIGNALS	_POSIX_VERSION

_POSIX_THREAD_PRIORITY_SCHEDULING

and defines the macros

_PC_ASYNC_IO	_SC_MQ_PRIO_MAX
_PC_CHOWN_RESTRICTED	_SC_NGROUPS_MAX
_PC_LINK_MAX	_SC_OPEN_MAX
_PC_MAX CANON	_SC_PAGE SIZE
_PC_MAX_INPUT	_SC_PRIORITIZED_IO
_PC_NAME_MAX	_SC_PRIORITY_SCHEDULING
_PC_NO_TRUNC	_SC_REALTIME_SIGNALS
_PC_PATH_MAX	_SC_RTSIG_MAX
_PC_PIPE_BUF	_SC_SAVED_IDS
_PC_PRIO_IO	_SC_SEMAPHORES
_PC_SYNC_IO	_SC_SEM_NSEMS_MAX
_PC_VDISABLE	_SC_SEM_VALUE_MAX
_SC_AIO_LISTIO_MAX	_SC_SHARED_MEMORY_OBJECTS
_SC_AIO_MAX	_SC_SIGQUEUE_MAX
_SC_AIO_PRIO_DELTA_MAX	_SC_STREAM_MAX
_SC_ARG_MAX	_SC_SYNCHRONIZED_IO
_SC_ASYNCHRONOUS_IO	_SC_THREADS
_SC_CHILD_MAX	_SC_THREAD_ATTR_STACKADDR
_SC_CLK_TCK	_SC_THREAD_ATTR_STACKSIZE
_SC_DELAYTIMER_MAX	_SC_THREAD_KEYS_MAX
_SC_FSYNC	_SC_THREAD_PRIO_INHERIT
_SC_GETGR_R_SIZE_MAX	_SC_THREAD_PRIO_PROTECT
_SC_GETPW_R_SIZE_MAX	_SC_THREAD_PROCESS_SHARED

_SC_JOB_CONTROL
_SC_LOGIN_NAME_MAX
_SC_MAPPED_FILES
_SC_MEMLOCK
_SC_MEMLOCK_RANGE
_SC_MEMORY_PROTECTION
_SC_MESSAGE_PASSING
_SC_MQ_OPEN_MAX
_SC_MQ_PRIO_MAX
_SC_THREAD_SAFE_FUNCTIONS
_SC_THREAD_STACK_MIN
_SC_THREAD_THREADS_MAX
_SC_TIMERS
_SC_TIMER_MAX
_SC_TTY_NAME_MAX
_SC_TZNAME_MAX
_SC_VERSION

_SC_THREAD_DESTRUCTOR_ITERATIO NS
_SC_THREAD_PRIORITY_SCHEDULING

and defines the types

*size_t** *ssize_t**

and declares the functions

_exit()
access()
alarm()
chdir()
chown()
close()
ctermid()
cuserid()
dup2()
dup()
execl()
execle()
execlp()
execv()
execve()
execvp()
fdatasync()
fork()
fpathconf()
fsync()
ftruncate()
getcwd()
getegid()
geteuid()
getgid()
getgroups()
getlogin()
getlogin_r()
getpgrp()
getpid()
getppid()
getuid()
isatty()
link()
lseek()
pathconf()
pause()
pipe()
read()
rmdir()
setgid()
setpgid()
setsid()
setuid()
sleep()
sysconf()
tcgetpgrp()
tcsetpgrp()
ttyname()
ttyname_r()
unlink()
write()

NOTE — The *cuserid*() symbol is permitted in this header, but need not be supplied. See 545 2.7.2.

<utime.h>

The header declares the structure

utimbuf

with structure elements

actime *modtime*

and the function

utime()

Annex D Profiles

(Informative)

This standard contains a number of options and variables that reflect the range of systems and environments that might be encountered. In general, it will be useful for applications to take the full range of these possibilities into account and either accommodate them or exclude them. However, there are significant communities of interest that may have common needs that warrant focusing on a specific suite of these options and parameters. This annex discusses the concept of *profiles* (also known as *functional standards*) and how they address this problem.

This annex reflects current thinking. It is clear that a concept such as this will help significantly in clarifying the intended use of these standards. It is to be expected that some of the details of this material will be changed before it is fully stabilized.

As background: the OSI model has over 170 standards (and consequent combinations thereof) that fit within it. Only a fraction of those are actually useful for any given application environment. The concept of *profiles* was developed to address this issue and appears also to apply to the area of application portability. The ISO/IEC term for such profiles is ISP, or "International Standardized Profile."

D.1 Definitions

The following definitions are proposed for use in the area covered by this part of ISO/IEC 9945.

D.1.1 Applications Environment Profile (AEP) [profile]:

The specification of a complete and coherent subset of an Open System Environment, together with the options and parameters necessary to support a class of applications for interoperability or applications portability, including consistency of data access and human interfaces. Where there are several AEPs for the same OSE, they are harmonized.

AEPs are the basis for procurement and conformance testing and are the target environment for software development.

D.1.2 Application Specific Environment (ASE):

A complete and coherent subset of an Applications Environment Profile, together with interfaces, services, or supporting formats outside of the profile, that are required by a particular application for its installation and execution.

D.1.3 Application Specific Environment Description (ASED):

The specification of an Application Specific Environment, together with the specific options or parameters required; interfaces, services, or supporting formats outside of the profile; and resource requirements necessary for the satisfactory operation of the application. (For example, storage and performance requirements.)

(This term is intended for use in Applications Conformance clauses found in profiles.)

D.1.4 coherent: The parts are logically connected. (For example, if both FORTRAN and COBOL are specified, whether they can share files is specified.)

D.1.5 complete: Having all the necessary parts. (For example, if COBOL and SQL are both specified, then there is a COBOL binding to SQL, or at least an explanation of why not.)

D.1.6 comprehensive: A sufficiently broad range of functionality is covered that the needs of most Applications Environment Profiles are met.

D.1.7 consistent: The parts of the Open System Environment do not inherently conflict with each other. This does not preclude options that conflict, as long as an Applications Environment Profile can select a set that does not conflict.

D.1.8 harmonized: Where same functionality is needed in several profiles, it appears identically in all of them.

D.1.9 Open System Environment (OSE): A comprehensive and consistent set of international information technology standards and functional standards (profiles) that specify interfaces, services, and supporting formats to accomplish interoperability and portability of applications, data, and people. These are based on International Standards (ISO, IEC, CCITT, ...)

D.1.10 POSIX Open System Environment: A comprehensive and consistent set of ISO/IEC, regional, and national information technology standards and functional standards (profiles) that specify interfaces, services, and supporting formats for interoperability and portability of applications, data, and people that are in accord with ISO/IEC 9945 (POSIX).

No single component of the OSE, including ISO/IEC 9945, is expected to be required in all such profiles.

D.2 Options in This Part of ISO/IEC 9945

In terms of this part of ISO/IEC 9945, there are a number of features that could be specified in a profile. This list includes:

- The options listed in 1.3.1.3.
- The limits in 2.8. Regarding the the C Language Limits for the type *char*, care should be taken that those limits are not for the POSIX.1 definition of *character*, but for the one in the C language. For the POSIX.1 definition of *character*, the following limits from the C Standard {2} could be specified as well: {MB_LEN_MAX} and {MB_CUR_MAX}.
- The flags in 2.9.4.
- Instances of the word “may” throughout the document. (Note that not all instances of “may” constitute behavior that could or should be considered appropriate for specification in a profile. Some reflect implementation variants that should not matter to applications.)
- Features that are specified in a generic way for broad portability of the standard, that might reasonably be constrained in the more limited context of a profile. For such features, the constraint shall be documented in a profile so that the effects of the constraint on the standard would be clarified.

D.3 Related Standards

The other POSIX standards (ISO/IEC 9945-2 {B36} , in particular) are expected to form a major part of the POSIX OSE. Other formal standards, such as those listed in A, are also expected to be part of such a document (in particular, the C Standard {2}.)

Standards such as other languages, SQL, graphics standards such as GKS, and networking standards are also probable candidates for inclusion in the POSIX OSE.

D.4 Related Activities

In many ways, the work of NIST (in terms of FIPS), OSF, UNIX International, and X/Open often act like early (but sophisticated) profiles or metaprofiles, specifying a range of standards from which true profiles could select. They collect together many standards, specify options, and specify the relationship between the parts. These activities go well beyond profiles, as they add specifications that are not formal standards to the suite as well. Often these additional specifications point to areas where formal standards are required.

D.5 Relationship to IEEE Std 1003.0-1995

The IEEE P1003.0 working group has written a *Guide to the POSIX Open System Environment*. This guide presents an overview of open system concepts and their applications. The guide provides information to persons evaluating systems based on the existence of, and interrelationship among, application software standards, with the objective of enabling application portability and system interoperability. This guide also presents a framework that identifies key information system interfaces involved in application portability and system interoperability, and describes the services offered across these interfaces. Standards or standards activities associated with the services are identified where they exist or are in progress.

Annex E Sample National Profile

(Informative)

One class of "community of interest" for which profiles (as discussed in D) are useful is specific countries, where the general characteristics warrant specific focus to serve the needs of users in those countries. Such needs lead to a number of implications concerning the options available within this part of ISO/IEC 9945 and may warrant specification of complementary standards as well.

The following is an example of a country's needs with respect to this part of ISO/IEC 9945 and how those needs relate to other international standards as well as national standards. The example provided is included here for informative purposes and is not a formal standard in the country in question. It is provided by the Danish Standards Association and is as accurate as possible with regards to Danish needs.[17] This example national profile is worded as if it were a national standard.

A subclass of conforming implementations can be identified that meet the requirements of a specific profile. By documenting these either in national standards, in a document similar to an ISO/IEC ISP (an International Standardized Profile), or in an informative annex (such as this), these can be referenced in a consistent manner.

E.1 (Example) Profile for Denmark

NOTE — This profile is chosen both for its instructive value by being a European profile and the generality in the provisions it makes, addressing most of the relevant points. It does claim to be correct for Denmark, and the style is what would be expected in a real ISP. A collection of real ISPs would be as useful, and work is underway collecting these.

This is the definition of the Danish Standards Association POSIX.1 profile. Information on the actual data for the locale and coded character set mapping definitions are under development as part of an informative annex in ISO/IEC 9945-2 `{B36}`.[18]

The subset of conforming implementations that provide the required characteristics below is referred to as conforming to the "Danish Standards Association (DS) Environment Profile" for this part of ISO/IEC 9945.

The profile specifies no options according to POSIX.1 section 2.9.3. For section 2.8.4 in the `<limits.h>` specification, the {_POSIX_TZNAME_MAX} value shall be 7.

E.1.1 Character Encoding

Any character encoding with the required repertoire of the POSIX profile plus the following repertoire shall be allowed.

A "character set description file," as described in ISO/IEC 9945-2, {B36} shall use the symbolic character names of the `ISO_10646` *charmap* file described in the ISO/IEC 9945-2 {B36} sample profile annex for the characters encoded in the character set.

For the Danish and Greenlandic languages, the following characters shall be present in addition to the repertoire required by the POSIX profile: `<ae>`, `<o/>`, `<aa>`, `<AE>`, `<O/>`, and `<AA>`. For the Faroese language, the following

[17]Further information may be obtained from the Danish Standards Association, Attn: S142u22A11 POSIX WG, Box 77, DK-2900 Hellerup, Denmark; FAX: +45 31 62 30 77; Email: `posix@itc.dk`
The data is also available electronically by anonymous FTAM or FTP at the site dkuug.dk in the directory isp, where some other example national profiles, locales, and *charmaps* may also be found. They are also available by an archive server reached at `archive@dkuug.dk`; use "`Subject: help`" for further information.
More complete examples of profiles are expected to be available in future revisions of this part of ISO/IEC 9945 and in other POSIX standards.
[18]The 9945-2 document, "POSIX.2," is currently in the state of a Committee Document (CD), to be approved as a Draft International Standard.

characters shall be present in addition to the required POSIX locale characters and Danish repertoire: `<a'>`, `<i'>`, `<o'>`, `<u'>`, `<y'>`, `<d>`, `<A'>`, `<I'>`, `<O'>`, `<U'>`, `<Y'>`, and `<D->`.

Recommended character sets are ISO 8859-1 {B34} or ISO 10646 {B37} . The **CHARSET** environment variable shall be used to specify the processing character set; for instance, `ISO_8859-1` or `ISO_10646`. This shall be used to select the encoded character-set-specific versions of the locale definitions. If no **CHARSET** variable is present, `ISO_8859-1` shall be assumed.

E.1.2 Character Encoding and Display

For terminal equipment not capable of generating or showing the processing character set, the character names defined in the current *charmap* file shall be used: characters in the *charmap* file having two-character names shall be specified by the two-character name preceded by the `<intro>` character, and characters having *charmap* names longer than two characters shall be specified by the character name preceded by the `<intro>` character and an `<underline>` and followed by an `<underline>`. In names longer than two characters, an `<intro>` character and an `<underline>` character in sequence shall signify a literal `<underline>` character part of the character name. Two `<intro>` characters in sequence shall signify one `<intro>` character, both in names and in the general stream.

For input, if the character name is undefined in the current *charmap* file, the data shall be left untouched (including the `<intro>` character) and the behavior is implementation defined.

E.1.3 Locale Definitions

The following guideline is used for specifying the locale identification string:[19]

`"%2.2s_%2.2s.%s,%s"`, <*language*>, <*territory*>, <*character-set*>, <*version*>

where <*language*> shall be taken from ISO 639 {B32} and <*territory*> shall be the two-letter country code of ISO 3166 {B33} , if possible. The <*language*> shall be specified with lowercase letters only, and the <*territory*> shall be specified in uppercase letters only. An optional <*character-set*> specification may follow after a `<period>` for the name of the character set; if just a numeric specification is present, this shall represent the number of the international standard describing the character set. If the <*character-set*> specification is not present, the encoded character set specific locale shall be determined by the **CHARSET** environment variable, and if this is unset or null, the encoding of ISO 8859-1 {B34} shall be assumed. A parameter specifying a <*version*> of the profile may be placed after the optional <*character-set*> specification, delimited by `<comma>`. This may be used to discriminate between different cultural needs; for instance, dictionary order versus a more systems-oriented collating order.

Following the above guidelines for locale names, the national Danish locale string shall be

```
da_DK
```

In the following, the **TZ** variable shall be specified according to the current official daylight-saving-time rules in Denmark. Since Daylight Saving Time is politically decided and thus changeable, this is only a recommendation.

The locale definition for Denmark shall be as follows:

```
LANG    da_DK
TZ      CET-1CET DST,M3.5.0/M9.5.0
```

The locale definition for the Faroe Islands shall be as follows:

```
LANG    fo_DK
```

[19]The guideline was inspired by the *X/Open Portability Guide* {B77} . It is presented in the file format notation used by ISO/IEC 9945-2 {B36} .

TZ `UTC0UTC DST,M3.5.0/M9.5.0`

The locale definition for Western Greenland shall be as follows:

LANG `kl_DK`
TZ `UTZ+3VTZ,M3.5.0/M9.5.0`

The locale definition for Eastern Greenland shall be as follows:

LANG `kl_DK`
TZ `VTZ+2WTZ,M3.5.0/M9.5.0`

For the Faroe Islands and Greenland, only the **LC_TIME** and **LC_MESSAGES** data are different from the Danish language specifications.

Annex F Portability Considerations

(Informative)

This annex contains information to satisfy the recommendations of the TSG-1 Final Report [B78] . The first clause describes perceived user requirements and the second indicates how the facilities of this part of ISO/IEC 9945 satisfy those requirements. The third clause offers guidance to writers of profiles on how the configurable options, limits, and optional behavior of this part of ISO/IEC 9945 should be cited in profiles.

F.1 User Requirements

This clause describes the user requirements as perceived by the developers of this part of ISO/IEC 9945. The primary source for these requirements was an analysis of historical practice in widespread use, as typified by the base documents listed in the introduction to this part of ISO/IEC 9945.

This standard addresses the needs of users requiring open systems solutions for source-code portability of applications. It currently addresses

- Multiprogramming and process management (creating processes, signaling, etc.)
- Access to files and directories in a hierarchy of file systems (opening, reading, writing, deleting files, etc.)
- Access to asynchronous communications ports and other special devices
- Access to information about other users of the system
- Facilities supporting applications requiring bounded (realtime) response

This standard provides C-language interfaces. Thus, it also addresses the following requirements

- Specific requirements of the C binding
- Interaction of the C binding with the underlying I/O system
- Interaction of the C internationalization capabilities with the environment

Extensions in many areas are being prepared, and this annex will be revised as these extensions are completed.

The requirements of users of this standard can be summarized as a single goal: application source portability. The requirements of the user are stated in terms of the requirements of portability of applications. This in turn becomes a requirement for a standardized set of syntax and semantics for operations commonly found on many operating systems.

The following subclauses list the perceived requirements for application portability.

F.1.1 Configuration Interrogation

An application must be able to determine whether and how certain optional features are provided and to identify the system upon which it is running, so that it may appropriately adapt to its environment.

F.1.2 Process Management

An application must be able to manage itself either as a single process or as multiple processes. Applications must be able to manage other processes when appropriate.

Applications must be able to identify, control, create, and delete processes, and there must be communication of information between processes and to and from the system.

Applications must be able to use multiple flows of control with a process (threads) and synchronize operations between these flows of control.

F.1.3 Access to Data

Application must be able to operate on the data stored on the system, access it, and transmit it to other applications. Information must have protection from unauthorized or accidental access or modification.

Applications must have sufficient information to adapt to varying behaviors of the system.

F.1.4 Access to the Environment

Applications must be able to access the external environment to communicate their input and results.

F.1.5 Access to Determinism and Performance Enhancements

Applications must have sufficient control of resource allocation to ensure the timeliness of interactions with external objects.

F.1.6 Operating System Dependent Profile

The capabilities of the operating system may make certain optional characteristics of the base language in effect no longer optional, and this should be specified.

F.1.7 I/O Interaction

The interaction between the C language I/O subsystem and the I/O subsystem of this part of ISO/IEC 9945 must be specified.

F.1.8 Internationalization Interaction

The effects of the environment of this part of ISO/IEC 9945 on the internationalization facilities of the C language must be specified.

F.1.9 C Language Extensions

Certain functions in the C language must be extended to support the additional capabilities provided by this part of ISO/IEC 9945.

F.1.10 Future Growth

These requirements must be met to be able to create any useful set of applications. It is recognized that many interesting classes of applications cannot be written using only services meeting these requirements. Significant additions to this standard are being developed, and future addenda and revisions will meet many of these additional requirements.

F.2 Portability Capabilities

This clause describes the significant portability capabilities of this part of ISO/IEC 9945 and indicates how the user requirements listed in F.1 are addressed. The capabilities are listed in the same format as the preceding user requirements; they are summarized in Table F.1.

F.2.1 Configuration Interrogation

The *uname*() operation provides basic identification of the system. The *sysconf*(), *pathconf*(), and *fpathconf*() functions provide means to interrogate the implementation to determine how to adapt to the environment in which it is running. These values can be either static (indicating that all instances of the implementation have the same value) or dynamic (indicating that different instances of the implementation have the different values, or that the value may vary for other reasons, such as reconfiguration).

Table F.1—Portability Capability Summary

Configuration Interrogation
Process Management
Access to Data
Access to the Environment
Access to Determinism and Performance Enhancements
Operating System Dependent Profile
I/O Interaction
Internationalization Interaction
C Language Extensions
Future Growth

Unsatisfied Requirements

None directly. However, as new areas are added, there will be a need for additional capability in this area.

F.2.2 Process Management

The *fork*() and *exec* functions provide for the creation of new processes or the insertion of new applications into existing processes. The *exit*() function and *abort*() function, defined by the C Standard {2}, allow for the termination of a process by itself. The *wait*() and *waitpid*() functions allow one process to deal with the the termination of another.

The *times*() function allows for basic measurement of times used by a process. Various functions, including *getpid*(), *getppid*(), *getuid*(), *geteuid*(), *getgid*(), *getegid*(), *setuid*(), *setgid*(), *setsid*(), *getlogin*(), *getpwnam*(), *getpwuid*(), *getgrnam*(), and *getgrgid*() provide for access to the identifiers of processes and the identifiers and names of owners of processes (and files).

The various functions operating on environment variables provide for communication of information (primarily user configurable defaults) from parent to child process.

The operations on the current working directory control and interrogate the directory from which relative file name searches start. The *umask*() function controls the default protections applied to files created by the process.

The *alarm*() and *sleep*() operations allow the process to suspend until a timer has expired or to be notified when a period of time has elapsed. The *time*() operation interrogates the current time and date.

The signal mechanism provides for communication of events either from other processes or from the environment to the application, and the means for the application to control the effect of these events. The mechanism provides for external termination of a process and for a process to suspend until an event occurs. The mechanism also provides for a value to be associated with an event.

The Job Control option provides a means to group processes and control them as groups, and to control their access to the interface between the user and the system (the “controlling terminal”). It also provides the means to suspend and resume processes.

The Process Scheduling option provides control of the scheduling and priority of a process.

The Message Passing option provides a means for inter-process communication involving small amounts of data.

The Memory Management facilities provide control of memory resources and for the sharing of memory.

The Threads facilities provide multiple flows of control with a process (threads), synchronization between threads, association of data with threads, and controlled cancellation of threads.

Unsatisfied Requirements

The following areas are currently under consideration: process resource limits, and checkpointing and restarting of processes.

F.2.3 Access to Data

The *open*(), *close*(), and *pipe*() functions provide for access to files and data. Such files may be classical files, interprocess data channels (pipes), or devices. Additional type of objects in the filesystem are permitted and are being contemplated for standardization.

The *dup*(), *dup2*(), *fcntl*(), *stat*(), *fstat*(), *access*(), *chmod*(), *fchmod*(), *chown*(), *ftruncate*(), and *utime*() functions allow for control and interrogation of file and file-related objects, and their ownership, protections, and timestamps.

The *read*(), *write*(), and *lseek*() functions provide for data transfer from the application to files (in all their forms).

The *mkdir*(), *rmdir*(), *link*(), *unlink*(), *rename*(), *opendir*(), *readdir*(), *rewinddir*(), and *closedir*() functions provide for a complete set of operations on directories. Directories can arbitrarily contain other directories, and a single file can be mentioned in more than one directory.

The file-locking mechanism provides for advisory locking (protection during transactions) of ranges of bytes (in effect, records) in a file.

The *pathconf*() and *fpathconf*() functions provide for inquiry as to the behavior of the system where variability is permitted. Since this can vary with the location of the file, the inquiry includes a proposed location.

The Synchronized Input and Output option provides for assured commitment of data to media.

The Asynchronous Input and Output option provides for initiation and control of asynchronous data transfers.

Unsatisfied Requirements

The following areas are currently under consideration: control of accessibility to file types (which may be remote) with reduced semantics.

F.2.4 Access to the Environment

The operations and types in Section 7 are provided for access to asynchronous serial devices. The primary intended use for these is the controlling terminal for the application (the interaction point between the user and the system). They are general enough to be used to control any asynchronous serial device. The interfaces are also general enough to be used with many other device types as a user interface when some emulation is provided.

Less detailed access is provided for other device types, but in many instances an application need not know whether an object in the file system is a device or a file to operate correctly.

Unsatisfied Requirements

Detailed control of common device classes, specifically magnetic tape, is not provided.

F.2.5 Bounded (Realtime) Response

The Realtime Signals Extension provides queued signals and the prioritization of the handling of signals. The SCHED_FIFO and SCHED_RR scheduling policies provide control over processor allocation. The Semaphores option provides high performance synchronization. The Memory Management functions provide memory locking for control of memory allocation, file mapping for high performance, and shared memory for high-performance interprocess communication. The Message Passing option provides for interprocess communication without being dependent on shared memory.

Unsatisfied Requirements

An interface to provide performance advice on file allocation and transfers is being developed.

F.2.6 Operating System Dependent Profile

This standard makes no distinction between text and binary files. The values of EXIT_SUCCESS and EXIT_FAILURE are further defined.

Unsatisfied Requirements

None known, but the C Standard {2} may contain some additional options that could be specified.

F.2.7 I/O Interaction

Section 8 defines how each of the C Standard {2} *stdio* functions interacts with the POSIX.1 operations, typically specifying the behavior in terms of POSIX.1 operations.

Unsatisfied Requirements

None.

F.2.8 Internationalization Interaction

The POSIX.1 environment operations provide a means to define the environment for *setlocale*() and time functions such as *ctime*(). These functions then become fully specified in the POSIX.1 environment. An additional function to set the time conversion is provided in *tzset*().

Unsatisfied Requirements

See F.2.10.

F.2.9 C Language Extensions

The *setjmp*() and *longjmp*() functions are not defined to be cognizant of the signal masks defined for POSIX.1. Functions *sigsetjmp*() and *siglongjmp*() are provided to fill this gap.

Unsatisfied Requirements

None.

F.2.10 Future Growth

It is arguable whether or not all functionality to support applications is potentially within the scope of this part of ISO/IEC 9945. As a simple matter of practicality, it cannot be. Areas such as general networking, graphics, application domain-specific functionality, windowing, and the like should be in unique standards. As such, they are properly "Unsatisfied Requirements" in terms of providing fully portable applications, but ones which are outside the scope of this standard.

However, certain broad areas that are applicable are currently under consideration.

Security: All the functionality provided in this standard is subject to additional constraints when high levels of security are required. Additional functionality and constraints are now being investigated.

Internationalization: Only a small fraction of the requirements for writing applications that operate properly across varying cultures have been met. Much of this belongs in the underlying language, but some properly belongs in this part of ISO/IEC 9945, once consensus on the specific solutions can be reached.

F.3 Profiling Considerations

This clause offers guidance to writers of profiles on how the configurable options, limits, and optional behavior of this part of ISO/IEC 9945 should be cited in profiles. Profile writers should consult the general guidance in POSIX.0 {B39} when writing POSIX Standardized Profiles.

The information in this clause is an inclusive list of the current features that should be considered by profile writers. Further subsetting of this part of ISO/IEC 9945, including the specification of behavior currently described as unspecified, undefined, implementation defined, or with the verbs "may" or "need not," violates the intent of the developers of this part of ISO/IEC 9945 and the guidelines of TR 10000-1 {B40} . Work is in progress to identify application requirements (for example, embedded realtime systems) for subgroupings of the features.

F.3.1 Configuration Options

The options to support the various configuration options are listed in 1.3.1.3. Profile writers should consult the following list and the comments concerning user requirements addressed by various POSIX.1 components in F.2.

{NGROUPS_MAX}

A nonzero value indicates that the implementation supports supplementary groups.

This option is needed where there is a large amount of shared use of files but where a certain amount of protection is needed. Many profiles[20] are known to require this option; it should only be required if needed, but it should never be prohibited.

{_POSIX_ASYNCHRONOUS_IO}

The system provides concurrent process execution and input and output transfers.

This option was created to support historical systems that did not provide the feature. It should only be required if needed, but it should never be prohibited.

[20]There are no formally approved profiles of this part of ISO/IEC 9945 at the time of publication; the reference here is to various profiles generated by private bodies or governments.

{_POSIX_CHOWN_RESTRICTED}

The system restricts the right to "give away" files to other users.

This option should be carefully investigated before it is required. Some applications expect that they can change the ownership of files in this way. It is provided where either security or system account requirements cause this ability to be a problem. It is also known to be specified in many profiles.

{_POSIX_FSYNC}

The system supports file synchronization requests.

This option was created to support historical systems that did not provide the feature. Applications that are expecting guaranteed completion of their input and output operations should require the {_POSIX_SYNC_IO} option. This option should never be prohibited.

{POSIX_JOB_CONTROL}

The system supports the optional job control facilities appearing in Section 7

The option was created primarily to support historical systems that did not provide the feature. Many existing profiles now require it; it should only be required if needed, but it should never be prohibited. Most applications that use it can run when it is not present, although with a degraded level of user convenience.

{_POSIX_MAPPED_FILES}

The system supports a the mapping of regular files into the process address space.

Both this option and the {_POSIX_SHARED_MEMORY_OBJECTS} option provide shared access to memory objects in the process address space. The interfaces defined under this option provide the functionality of existing practice for mapping regular files. This functionality was deemed unnecessary, if not inappropriate, for embedded systems applications and, hence, is provided under this option. It should only be required if needed, but it should never be prohibited.

{_POSIX_MEMLOCK}

The system supports the locking of the address space.

This option was created to support historical systems that did not provide the feature. It should only be required if needed, but it should never be prohibited.

{_POSIX_MEMLOCK_RANGE}

The system supports the locking of specific ranges of the address space.

For applications that have well-defined sections that need to be locked and others that do not, the standard supports an optional set of interfaces to lock or unlock a range of process addresses. The following are two reasons for having a means to lock down a specific range:

- An asynchronous event handler function that must respond to external events in a deterministic manner such that page faults cannot be tolerated.
- An input/output "buffer" area that is the target for direct-to-process I/O, and the overhead of implicit locking and unlocking for each I/O call cannot be tolerated.

It should only be required if needed, but it should never be prohibited.

{_POSIX_MEMORY_PROTECTION}

The system supports memory protection.

The provision of this option typically imposes additional hardware requirements. It should never be prohibited.

{_POSIX_PRIORITIZED_IO}

The system provides prioritization for input and output operations.

The use of this option may interfere with the ability of the system to optimize input and output throughput. It should only be required if needed, but it should never be prohibited.

{_POSIX_MESSAGE_PASSING}

The system supports the passing of messages between processes.

This option was created to support historical systems that did not provide the feature. The functionality adds a high-performance interprocess communication facility for local communication. It should only be required if needed, but it should never be prohibited.

{_POSIX_PRIORITY_SCHEDULING}

The system provides priority-based process scheduling.

Support of this option provides predictable scheduling behavior, allowing applications to determine the order in which processes that are ready to run are granted access to a processor. It should only be required if needed, but it should never be prohibited.

{_POSIX_REALTIME_SIGNALS}

The system provides prioritized, queued signals with associated data values.

This option was created to support historical systems that did not provide the features. It should only be required if needed, but it should never be prohibited.

{_POSIX_SAVED_IDS}

The option was created primarily to support historical systems that did not provide the feature (typically the complement of the ones that at the time provided {_POSIX_JOB_CONTROL}). Many existing profiles now require it; it should only be required if needed, but it should never be prohibited. Certain classes of applications rely on it for proper operation, and there is no alternative short of giving the application “root” privileges on most implementations that do not provide {_POSIX_SAVED_IDS}.

{_POSIX_SEMAPHORES}

The system provides counting semaphores.

This option was created to support historical systems that did not provide the feature. It should only be required if needed, but it should never be prohibited.

{_POSIX_SHARED_MEMORY_OBJECTS}

The system supports the mapping of shared memory objects into the process address space.

Both this option and the {_POSIX_MAPPED_FILES} option provide shared access to memory objects in the process address space. The interfaces defined under this option provide the functionality of existing practice for shared memory objects. This functionality was deemed appropriate for embedded systems applications and, hence, is provided under this option. It should only be required if needed, but it should never be prohibited.

{_POSIX_SYNCHRONIZED_IO}

The system supports guaranteed file synchronization.

This option was created to support historical systems that did not provide the feature. Applications that are expecting guaranteed completion of their input and output operations should require this option, rather than the {_POSIX_FSYNC} option. It should only be required if needed, but it should never be prohibited.

{_POSIX_THREADS}

The system supports multiple threads of control within a single process.

This option was created to support historical systems that did not provide the feature. Applications written assuming a multithreaded environment would be expected to require this option. It should only be required if needed, but it should never be prohibited.

{_POSIX_THREADS_ATTR_STACKADDR}

The system supports specification of the stack address for a created thread.

Applications may take advantage of support of this option for performance benefits, but dependence on this feature should be minimized. This option should never be prohibited.

{_POSIX_THREADS_ATTR_STACKSIZE}

The system supports specification of the stack size for a created thread.

Applications may require this option in order to ensure proper execution, but such usage limits portability and dependence on this feature should be minimized. It should only be required if needed, but it should never be prohibited.

{POSIX_THREADS_PRIORITY_SCHEDULING}

The system provides priority-based thread scheduling.

Support of this option provides predictable scheduling behavior, allowing applications to determine the order in which threads that are ready to run are granted access to a processor. It should only be required if needed, but it should never be prohibited.

{_POSIX_THREADS_PRIO_INHERIT}

The system provides mutual exclusion operations with priority inheritance.

Support of this option provides predictable scheduling behavior, allowing applications to determine the order in which threads that are ready to run are granted access to a processor. It should only be required if needed, but it should never be prohibited.

{_POSIX_THREADS_PRIO_PROTECT}

The system supports a priority ceiling emulation protocol for mutual exclusion operations.

Support of this option provides predictable scheduling behavior, allowing applications to determine the order in which threads that are ready to run are granted access to a processor. It should only be required if needed, but it should never be prohibited.

{_POSIX_THREADS_PROCESS_SHARED}

The system provides shared access among multiple processes to synchronization objects.

This option was created to support historical systems that did not provide the feature. It should only be required if needed, but it should never be prohibited.

{_POSIX_THREAD_SAFE_FUNCTIONS}

The system provides thread-safe versions of all of the POSIX.1 functionality.

This option is required if the {_POSIX_THREADS} option is supported. This is a separate option because thread-safe functions are useful in implementations providing other mechanisms for concurrency. It should only be required if needed, but it should never be prohibited.

{_POSIX_TIMERS}

The system provides higher resolution clocks with multiple timers per process.

This option was created to support historical systems that did not provide the features. This option is appropriate for applications requiring higher resolution timestamps or needing to control the timing of multiple activities. It should only be required if needed, but it should never be prohibited.

F.3.2 Configurable Limits

In general, the configurable limits in 2.8 have been set to minimal values; many applications or implementations may require larger values. No profile can cite lower values.

{AIO_LISTIO_MAX}

The current minimum is likely to be inadequate for most applications. It is expected that this value will be increased by profiles requiring support for list input and output operations.

{AIO_MAX}

The current minimum is likely to be inadequate for most applications. It is expected that this value will be increased by profiles requiring support for asynchronous input and output operations.

{AIO_PRIO_DELTA_MAX}

The functionality associated with this limit is needed only by sophisticated applications. It is not expected that this limit would need to be increased under a general-purpose profile.

{ARG_MAX}

The current minimum is likely to need to be increased for profiles, particularly as larger amounts of information are passed through the environment. Many implementations are believed to support larger values.

{CHILD_MAX}

The current minimum is suitable only for systems where a single user will not be running applications in parallel. It is significantly too low for any system also requiring windows, and if {_POSIX_JOB_CONTROL} is specified, it should be raised.

{CLOCKRES_MIN}

It is expected that profiles will require a finer granularity clock, perhaps as fine as 1 μs, represented by a value of 1000 for this limit.

{DELAYTIMER_MAX}

It is believed that most implementations will provide larger values.

{LINK_MAX}

For most applications and usage, the current minimum is adequate. Many implementations have a much larger value, but this should not be used as a basis for raising the value unless the applications to be used will require it.

{LOGIN_NAME_MAX}

This is not actually a limit, but an implementation parameter. No profile should impose a requirement on this value.

{MAX_CANON}

For most purposes, the current minimum is adequate. Unless high-speed burst serial devices are to be used, it should be left as is.

{MAX_INPUT}

See {MAX_CANON}.

{MQ_OPEN_MAX}

The current minimum should be adequate for most profiles.

{MQ_PRIO_MAX}

The current minimum corresponds to the required number of process scheduling priorities. Many realtime practitioners believe that the number of message priority levels ought to be the same as the number of execution scheduling priorities.

{NAME_MAX}

Many implementations now support larger values, and many applications and users assume that larger names can be used. Many existing profiles also specify a larger value. Specifying this value will reduce the number of conforming implementations, although this may not be significant consideration over time. Values greater than 255 should not be required.

{NGROUPS_MAX}

Nonzero values act as an option (as discussed in F.3.1). The value selected might be typically 8 or larger.

{OPEN_MAX}

The historically common value for this has been 20. Many implementations support values larger than that. If applications that use larger values are anticipated, they should be specified.

{PAGESIZE}

This is not actually a limit, but an implementation parameter. No profile should impose a requirement on this value.

{PATH_MAX}

Historically, the minimum has been either 1024 or indefinite, depending on the implementation. Few applications actually require values larger than 256, but some users might create file hierarchies that must be accessed with longer paths. This value should only be changed if there is a clear requirement.

{PIPE_BUF}

The current minimum is adequate for most applications. Historically, it has been larger. If applications that write single transactions larger than this are anticipated, it should be increased. Applications that write lines of text larger than this probably do not need it increased, as the text line will be delimited by a newline.

{POSIX_VERSION}

This is actually not a limit, but a standard version stamp. Generally, a profile should specify this standard by a name in the normative references section, not this value.

{PTHREAD_DESTRUCTOR_ITERATIONS}

It is unlikely that applications will need larger values to avoid loss of memory resources.

{PTHREAD_KEYS_MAX}

The current value should be adequate for most profiles.

{PTHREAD_STACK_MIN}

This should not be treated as an actual limit, but as an implementation parameter. No profile should impose a requirement on this value.

{PTHREAD_THREADS_MAX}

It is believed that most implementations will provide larger values.

{RTSIG_MAX}

The current limit was chosen so that the set of POSIX.1 signal numbers can fit within a 32 b field. It is recognized that most existing implementations define many more signals than are specified in POSIX.1 and, in fact, many implementations have already exceeded 32 signals (including the "null signal"). Support of {_POSIX_RTSIG_MAX} additional signals may push some implementation over the single 32 b word line, but is unlikely to push any implementations that are already over that line beyond the 64 signal line.

{SEM_NSEMS_MAX}

The current value should be adequate for most profiles.

{SEM_VALUE_MAX}

The current value should be adequate for most profiles.

{SSIZE_MAX}

This limit reflects fundamental hardware characteristics (the size of an integer) and should not be specified unless it is clearly required. Extreme care should be taken to assure that any value that might be specified does not unnecessarily eliminate implementations because of accidents of hardware design.

{STREAM_MAX}

This limit is very closely related to {OPEN_MAX}. It should never be larger than {OPEN_MAX}, but could reasonably be smaller for application areas where most files are not accessed through *stdio*. Some implementations may limit {STREAM_MAX} to 20 but allow {OPEN_MAX} to be considerably larger. Such implementations should be allowed for if the applications permit.

{TIMER_MAX}

The current limit should be adequate for most profiles, but it may need to be larger for applications with a large number of asynchronous operations.

{TTY_NAME_MAX}

This is not actually a limit, but an implementation parameter. No profile should impose a requirement on this value.

{TZNAME_MAX}

The minimum has been historically adequate, but if longer timezone names are anticipated (particularly such values as "UTC-1"), this should be increased.

F.3.3 Optional Behavior

In this part of ISO/IEC 9945, there are no instances of the terms unspecified, undefined, implementation defined, or with the verbs "may" or "need not," that the developers of this part of ISO/IEC 9945 anticipate or sanction as suitable for profile or test method citation. All of these are merely warnings to portable applications to avoid certain areas that can vary from system to system, and even over time on the same system. In many cases, these terms are used explicitly to support extensions, but profiles should not anticipate and require such extensions; future versions of the standard may do so.

Annex G Performance Metrics

(Informative)

Performance metrics are an essential element in evaluation and use of a system. To this end, for each facility, a set of performance metrics was defined to assist in uniform treatment of the measurement of performance between different conforming implementations. Moreover, in order to achieve full functionality, a realtime application may have requirements on the performance of an implementation according to the performance metrics defined by this annex.

This annex contains the results of the performance metrics work. It is included here as a guide to what the performance expectations might be for the interfaces defined in this standard.

G.1 Performance Measurement Documentation

An implementation may optionally provide performance documentation. An implementation claiming conformance to this part of ISO/IEC 9945 with the POSIX Realtime Performance Documentation option shall provide a document with measurements of the performance metrics for the required functionality of this part of ISO/IEC 9945 and the metrics associated with each of the options of this part of ISO/IEC 9945 that it supports. For each metric, the documentation shall include the mean observed value and the worst-case guaranteed behavior, together with the condition under which the worst-case behavior is incurred. The performance measurement shall be cross-referenced to the standard and section number defining the metric or function.

This document shall provide performance measurements for the following function calls defined in this part of ISO/IEC 9945:

```
calloc                     longjmp                    siglongjmp
fcntl (fd, F_GETLK, ...)   malloc                     sigpending
fcntl (fd, F_SETLK, ...)   read                       sigprocmask
fpathconf                  realloc                    sigsetjmp
free                       setjmp                     sysconf
fstat                      sigaction                  write
kill
```

For all documented metrics, the conditions of measurement shall be supplied. Conditions of measurement is defined as the methodology used, software and hardware configuration, call parameters, system loading, initial conditions, and any other information required to duplicate the measurement. Where a measurement varies due to differences in call parameters or initial conditions, multiple measurements may be provided. Multiple measurements of a given metric shall be accompanied by the altered conditions of measurement. Where measurements may deviate from the documented value, the conditions of deviation shall be supplied along with maximum deviation possible. Conditions of deviation should include at least the following where applicable:

1) Interrupts
2) Cache residence status
3) Memory conflicts
4) Bus conflicts
5) Virtual memory paging
6) Other hardware resource conflicts

G.2 Signals

Because realtime applications require fast and deterministic response to both external and internal events, it is important that the system provide accurately characterized event notification performance. Each of the metrics below assumes that the user application is the highest priority user process in the system. Worst-case observed measurements should be provided.

If the implementation supports both the Realtime Signals Extension and Realtime Performance Documentation options, it shall report

Event Dispatch Latency

This is the time interval between the occurrence of an event (generation of the signal) and the execution of the first instruction of the signal-catching function in response to the event.

For the purposes of this metric, an event is defined as one resulting from the execution of the *sigqueue*() function with the pid of a higher priority process.

NOTE — Realtime applications that respond to sporadic external events or that use a timer-driven event handler to implement a periodic realtime application need to know how long it takes to enter the application event handler in response to the event. This is one of the primary performance metrics used to compare existing realtime kernels and operating systems.

Signal Waiting Overhead With an Indefinite Wait

This is the time interval between the initiation of a *sigwaitinfo*() function and the completion of that function. It is measured when a blocked signal is polled that currently has at least one queued signal pending.

Signal Waiting Overhead With a NonZero Timeout Period

This is the time interval between the initiation of a *sigtimedwait*() function and the completion of that function. It is measured when a blocked signal is polled that currently has at least one queued signal pending and a nonzero timeout period is specified.

Signal Waiting Overhead With a Zero Timeout Period

This is the time interval between the initiation of a *sigtimedwait*() function and the completion of that function. It is measured when a blocked signal is polled that currently has at least one queued signal pending and a zero timeout period is specified.

Signal Waiting Overhead With No Pending Signal

This is the time interval between the initiation of a *sigtimedwait*() function and the completion of that function. It is measured when a blocked signal is polled that does not have any queued signals pending and a zero timeout period is specified.

NOTE — For the same classes of applications that are implemented as polling loops instead of interrupt handlers, the overhead of polling is the metric of interest. The metric is required both with and without timeout so that the application designer can know the cost of the various features.

G.3 Synchronized Input and Output

The purpose of these metrics is twofold:

1) To allow the user to compare the performance of synchronized I/O on equivalent hardware platforms, and
2) On a given platform, to distinguish the following performance differences.
   a) Regular I/O versus Synchronized I/O
   b) Synchronized I/O with data integrity versus file integrity

G.3.1 Conformance

If the implementation supports both the Synchronized Input and Output and Realtime Performance Documentation options, it shall report, for at least one hardware platform specified by the vendor, all metrics described in this subclause. A conforming implementation may specify any given metric as “Not Applicable” if the system under test does not support the function tested or if use of that function has no visible effect on file I/O performance.

G.3.2 Metrics

The vendor shall first specify a number, referred to as the **Fundamental Transfer Size**, which, for rotating magnetic media, should correspond to the block size for the device. If this value is other than the block size, the vendor shall document the significance of the value chosen.

G.3.3 Transfer Metrics

For each of the metrics defined, the vendor shall specify multiple transfer timings. These timings shall be taken under the conditions defined for each metric under each of five different pairs of file size and values of *nbyte*. These pairs are:

File Size (in bytes)	***nbytes***
65 536	Fundamental transfer size / 2
65 536	Fundamental transfer size
65 536	Optimal transfer size
2 097 152	Fundamental transfer size
2 097 152	Optimal transfer size

If the values 65 536 and 2 097 152 are not integral multiples of the *Fundamental Transfer Size*, the vendor may select the integral multiples of the *Fundamental Transfer Size* that are immediately above these values and shall specify the values used.

If the largest possible file size supported by the implementation is smaller than 65 536 B, the vendor shall provide and document metrics for the largest possible size and approximately 1/32 of that size. If the largest possible file size supported by the implementation is smaller than 2 097 152 B but larger than 65 536 B, the vendor shall provide and document metrics for the largest possible size, for approximately 1/32 of that size, and for 65 536 B. The vendor shall specify the size used.

For those metrics that include actual I/O, the implementor shall identify that portion of the metric that is contributed by the I/O device to transfer the data.

Open File Time, Unsynchronized I/O

This metric measures the time to open a file without synchronized I/O.

Open File Time, Synchronized I/O, Data Integrity

This metric measures the time to open a file with synchronized I/O data integrity (O_DSYNC).

Open File Time, Synchronized I/O, File Integrity

This metric measures the time to open a file with synchronized I/O file integrity (O_SYNC).

Unsynchronized Input Time

This metric measures the time to input data from a file using normal I/O. This metric forms a base of comparison for the synchronized input operation metrics. This metric shall be taken for each possible combination of file size and transfer size.

Unsynchronized Output Time

This metric measures the time to output data to a file using normal I/O. This metric forms a base of comparison for the synchronized output operation metrics. This metric shall be taken for each possible combination of file size and transfer size.

Synchronized Input Time, Data Integrity

This metric measures the time to input data from a file using synchronized I/O with data integrity (O_DSYNC|O_RSYNC). This metric shall be taken for each possible combination of file size and transfer size.

Synchronized Input Time, File Integrity

This metric measures the time to input data from a file using synchronized I/O with file integrity (O_SYNC|O_RSYNC). This metric shall be taken for each possible combination of file size and transfer size.

Synchronized Output Time, Data Integrity

This metric measures the time to output data to a file using synchronized I/O with data integrity (O_DSYNC). This metric shall be taken for each possible combination of file size and transfer size.

Synchronized Output Time, File Integrity

This metric measures the time to output data to a file using synchronized I/O with file integrity (O_SYNC). This metric shall be taken for each possible combination of file size and transfer size.

NOTE — It is recognized that the area of standardizing metrics for file I/O is nebulous. What is presented here is an overall set of metrics for the total Synchronized I/O model presented in this section. Individual vendors will find it necessary to interpret this section with regard to each different hardware platform. This interpretation should be consistent with the purpose of the metrics.

It became apparent that the performance measurement of *fsync*() and *fdatasync*() was not realistically possible. There is no means, short of modifying the I/O subsystem of the operating system, of obtaining meaningful measurements. As writes are performed, they are also being updated to disk asynchronously, consequently making it impossible to control the number of disk blocks that will actually be updated.

G.4 Asynchronous Input and Output

If the implementation supports both the Asynchronous Input and Output and Realtime Performance Documentation options, it shall report

Asynchronous I/O Request Time, No Signal, aio_read

The time required to queue an *aio_read*() request and return control to the requesting process. For this measurement, the *aio_sigevent.sigev_signo* member shall be zero.

Asynchronous I/O Request Time, No Signal, aio_write

The time required to queue an *aio_write*() request and return control to the requesting process. For this measurement, the *aio_sigevent.sigev_signo* member shall be zero.

Asynchronous I/O Request Time, No Signal, aio_fsync

The time required to queue an *aio_fsync*() request and return control to the requesting process. For this measurement, the *aio_sigevent.sigev_signo* member shall be zero.

Asynchronous I/O Request Time, Regular Signal, aio_read

The time required to queue an *aio_read*() request and return control to the requesting process. For this measurement, the *aio_sigevent.sigev_signo* member shall be equal to SIGUSR1.

Asynchronous I/O Request Time, Regular Signal, aio_write

The time required to queue an *aio_write*() request and return control to the requesting process. For this measurement, the *aio_sigevent.sigev_signo* member shall be SIGUSR1.

Asynchronous I/O Request Time, Regular Signal, aio_fsync

The time required to queue an *aio_fsync*() request and return control to the requesting process. For this measurement, the *aio_sigevent.sigev_signo* member shall be equal to SIGUSR1.

Asynchronous I/O Request Time, No Signal, lio_listio

The time required to queue a *lio_listio*() request and return control to the requesting process. The *mode* shall be set to LIO_NOWAIT, *sig* shall be **NULL**, and there shall be a single valid entry in *list*. The entry shall have *aio_lio_opcode* equal to LIO_READ.

Asynchronous I/O Request Time, Regular Signal, lio_listio

The time required to queue a *lio_listio*() request and return control to the requesting process. The *mode* shall be set to LIO_NOWAIT, the *sig* argument shall refer to a *sigevent* structure with *sigev_signo* equal to SIGUSR1, and there shall be a single valid entry in *list*. The entry shall have *aio_lio_opcode* equal to LIO_READ.

Signal Delivery Latency, Regular Signal

The time from an asynchronous I/O completion to the first execution executed within the process signal handler for the signal SIGUSR1.

Signal Delivery Latency, Iosuspend, No Completed I/O

The time from an asynchronous I/O completion to when the process returns from a call to *aio_suspend*() with a single *aiocb* as argument.

Signal Delivery Latency, Sigwaitinfo, No Completed I/O, Regular Signal

The time from an asynchronous I/O completion to when the process returns from a call to *sigwaitinfo*() that synchronously receives the signal. The signal used for this measurement shall be SIGUSR1.

If the implementation supports all of the Asynchronous Input and Output, Realtime Signals Extension, and Realtime Performance Documentation options, it shall report

Asynchronous I/O Request Time, Realtime Signal, aio_read

The time required to queue an *aio_read*() request and return control to the requesting process. For this measurement, the *aio_sigevent.sigev_signo* member shall be equal to a number between SIGRTMIN and SIGRTMAX.

Asynchronous I/O Request Time, Realtime Signal, aio_write

The time required to queue an *aio_write*() request and return control to the requesting process. For this measurement, the *aio_sigevent.sigev_signo* member shall be equal to a number between SIGRTMIN and SIGRTMAX.

Asynchronous I/O Request Time, Realtime Signal, aio_fsync

The time required to queue an *aio_fsync*() request and return control to the requesting process. For this measurement, the *aio_sigevent.sigev_signo* member shall be equal to a number between SIGRTMIN and SIGRTMAX.

Asynchronous I/O Request Time, Realtime Signal, lio_listio

The time required to queue a *lio_listio*() request and return control to the requesting process. For this measurement, *mode* shall be set to LIO_NOWAIT, the *sig* argument shall refer to a *sigevent* structure with *sigev_signo* equal to a number between SIGRTMIN and SIGRTMAX, and there shall be a single valid entry in *list*. The entry shall have *aio_lio_opcode* equal to LIO_READ.

Signal Delivery Latency, Realtime Signal

The time from an asynchronous I/O completion to the first execution executed within the process signal handler for a realtime extended signal. See G.2.

Signal Delivery Latency, Sigwaitinfo, No Completed I/O, Realtime Signal

The time from an asynchronous I/O completion to when the process returns from a call to *sigwaitinfo*() that synchronously receives the signal. The signal used for this measurement shall be in the range of SIGRTMIN to SIGRTMAX.

G.5 Semaphores

The usability of a realtime system for a given application depends on the speed with which the system can effectively synchronize processes. This, in turn, depends on the cost of locking and posting to a semaphore.

If the implementation supports both the Semaphores and Realtime Performance Documentation options, it shall report

Semaphore Unconditional Unlocking Time, No Waiters

This is the time required by the *sem_post*() function when there is no process waiting for the semaphore.

Semaphore Unconditional Unlocking Time, No Switch Required

This is the time required by the *sem_post*() function when there is a process waiting for the semaphore, but the process does not preempt the process calling *sem_post*().

Semaphore Unconditional Unlocking Time, Switch Required

This is the time required by the *sem_post*() function when there is a process waiting for the semaphore, and the process preempts the process calling *sem_post*(). The time reported shall not include time spent by the preempting process.

Semaphore Unconditional Locking Time, Semaphore Unlocked

This is the time required by the *sem_wait*() function when the semaphore is unlocked prior to calling *sem_wait*().

Semaphore Unconditional Locking Time, Semaphore Locked

This is the time required by the *sem_wait*() function when the semaphore is locked prior to calling *sem_wait*(). The time measured shall exclude all time the process is blocked while waiting for another process to unlock the semaphore. The time measured shall exclude all time taken by a process other than the one performing the *sem_wait*() function.

Semaphore Conditional Locking Time, Semaphore Unlocked

This is the time required by the *sem_trywait*() function when the semaphore is unlocked prior to calling *sem_trywait*().

Semaphore Conditional Locking Time, Semaphore Locked

This is the time required by the *sem_trywait*() function when the semaphore is locked prior to calling *sem_trywait*().

G.6 Mutexes and Condition Variables

If the implementation supports both the Threads and Realtime Performance Documentation options, it shall report

Mutex Lock/Unlock with No Contention

This is the time interval needed to call *pthread_mutex_lock*() followed immediately by *pthread_mutex_unlock*() on a mutex that is unowned and that is only being used by the thread doing the test.

Mutex Lock/Unlock with Contention

This is the time interval between when one thread calls *pthread_mutex_unlock*() and another thread that was blocked on *pthread_mutex_lock*() returns with the lock held.

On a machine with *p* processors, this metric should be provided for a system dedicated to running a single instance of this test and a system with *2*p* processes, each running an instance of this test in a tight loop.

Condition Variable Signal/Broadcast with No Waiters

This is the amount of time needed to execute *pthread_cond_signal*() or *pthread_cond_broadcast*() if there are no threads blocked on the condition.

Condition Variable Wake Up

This is the amount of time from when one thread calls *pthread_cond_signal*() and a thread blocked on that condition variable returns from its *pthread_cond_wait*() call. The condition and its associated mutex should not be used by any other thread. Metrics shall be provided for both the case when the *pthread_cond_signal*() call is executed under the associated mutex, as well as not under the mutex. In addition, on a machine with p processors, the metric shall be provided for a system dedicated to running a single instance of this test and a system with 2**p* processes, each running an instance of this test in a tight loop. The scheduling parameters used (such as the scheduling policy and priority for each thread) have to be specified along with this metric.

Time of Wakeup After Timed Wait

This is the time required for the highest priority thread to resume execution after a call to *pthread_cond_timedwait*(). Metrics shall be provided for both the case when the *pthread_cond_timedwait*() call is awakened by a call to *pthread_cond_signal*() and when the absolute time to be awaited has already been passed at the time of the call.

G.7 Process Memory Locking

If the implementation supports both the Process Memory Locking and Realtime Performance Documentation options, it shall report

Maximum Locked Memory Access Time

This metric measures the upper bound on the amount of time required for an application to fetch data of size *int* from locked memory. The measurement shall be made by fetching from and storing to a C language "static" variable allocated within a range locked by *mlock*() or *mlockall*(). The implementor shall show in the metrics documentation the C code used to perform the measurement and document other conditions of measurement, such as process priority, special memory types, etc., necessary to obtain equivalent measurements.

G.8 Shared Memory

If the implementation supports both the Shared Memory and the Realtime Performance Documentation options, and it supports either of the Process Memory Locking or Range Memory Locking options, it shall report whether using locked, shared memory objects may cause I/O activity.

In implementations that support shared memory object through mapped files, it is important to document whether using shared memory may result in some I/O traffic.

G.9 Execution Scheduling

The following measures are defined assuming that the priority setting operations are performed by the highest priority process in the system (i.e., the running process on a uniprocessor) under system workload conditions indicated for that measure. The test environment shall consist of a *priority_setting* process and, in some cases, a *target* process that shall perform the tests and zero or more background workload processes. These measures are selected due to their importance in predicting application performance. The required metrics, the measurement environment, required measurements, measurement methodology, and other requirements placed on the implementation are further defined below.

The metrics shall be reported for both SCHED_RR and SCHED_FIFO scheduling policies.

G.9.1 Measurement Environment

The test environment for these measurements shall consist of background workloads containing the following number of processes in addition to the priority_setting and target processes:

1) None
2) One
3) MAX_PROC/2, where MAX_PROC is the implementation-defined maximum number of processes that may be present on the system at any one time
4) MAX_PROC-2
5) If MAX_PROC > 32, 33 plus the number of processors minus 1
6) If MAX_PROC > 64, 65 plus the number of processors minus 1
7) If MAX_PROC > 128, 129 plus the number of processors minus 1

In every case, the number of processes shall not exceed MAX_PROC-2, regardless of the number of processors.

Each background workload process except the process that may become runnable as a result of the priority setting operation shall be so constructed that it is always a runnable process (i.e., all such processes shall be compute-bound.)

Priorities of the workload processes shall be uniformly distributed across the allowable range of priorities, subject to the constraint that they shall not exceed the measured process or its target.

G.9.2 Metrics Definitions

If the implementation supports both the Process Scheduling and Realtime Performance Documentation options, it shall report

Set Self Same

The priority_setting process shall set its priority to its current priority value. The priority_setting process shall continue execution. The measurement shall be made by taking the difference between a time-stamp taken immediately prior to the *sched_setparam*() call and a time-stamp taken immediately after return from the call.

Set Self Lower

The priority_setting process shall set its priority to a value which shall cause it to be replaced as the running process by the target process [using *sched_setparam*() or *sched_yield*()]. The measurement shall be made by taking the difference between a time-stamp taken immediately prior to the *sched_setparam*() call and a time-stamp taken immediately upon execution of the target process.

Set Target Same

The priority_setting process shall set the priority of the target process to its current priority value. The priority_setting process shall continue execution. The measurement shall be made by taking the difference between a time-stamp taken immediately prior to the *sched_setparam*() call and a time-stamp taken immediately after return from the call.

Set Target Higher

The priority_setting process shall set the priority of the target process to a value that shall cause the priority_setting process to be replaced as the running process by the target process. The measurement shall be made by taking the difference between a time-stamp taken immediately prior to the *sched_setparam*() call and a time-stamp taken immediately upon execution of the target process.

SCHED_RR rr_interval Accuracy

In an environment with some number of processes (more than one) at a single priority, all using the SCHED_RR scheduling policy, the implementation shall measure the deviation of the time at which the processes are moved to the tail of their priority queues relative to the value returned by *sched_rr_get_interval*() for those processes. This deviation shall be reported as a mean, maximum observed, and minimum observed values.

If the implementation supports the {_POSIX_THREADS} option, it shall report

Thread Yield Time (Busy)

This is the amount of time between that point when a running thread voluntarily gives up the CPU until the highest priority runnable thread begins execution of its application code.

Scheduler Overhead

This is the amount of time needed to effect fair scheduling policies for timesharing threads by having the implementation interrupt the current executing thread at regular intervals to recalculate priorities and gather statistics. In an ideal realtime environment, threads would be immune from such overhead but, if not, this overhead needs to be measured. The values to be reported shall be the highest rate at which the implementation causes application tasks to be preempted for managing scheduling parameters, and the maximum time such preemption can last, along with a description of the functionality provided at such preemption points.

Scheduling Attribute Change Time (No Context Switch)

This is the time needed to execute a *pthread_getschedparam*() and *pthread_setschedparam*() pair when the result of the set does not change which thread is currently running.

Scheduling Attribute Change Time (Context Switch)

This is the amount of time between that point when a running thread begins a *pthread_attr_getschedparam*() and *pthread_attr_setschedparam*() pair that alters the running thread or some other thread in such a way that the setting thread relinquishes the CPU to another thread, and until that other thread begins execution of its application code.

If the symbol {_POSIX_THREAD_PRIO_INHERIT} is defined, the performance metrics *Mutex Lock/Unlock, No Contention and Mutex Lock/Unlock, Contention*, should be given for both the cases when the protocol attribute is set to PTHREAD_PRIO_NONE or PTHREAD_PRIO_INHERIT. If the symbol{_POSIX_THREAD_PRIO_PROTECT} is defined, these metrics should be given for both the cases when the protocol attribute is set to PTHREAD_PRIO_NONE or PTHREAD_PRIO_PROTECT.

G.9.3 Other Requirements

For those tests that require the target process to run and take a time-stamp, the target process shall be constructed so that it makes the priority-setting process a process that is able to run via a *sched_setparam*() function call and performs the time-stamp immediately following that call. The priority of the target process and the priority-setting process shall be such that the target process is the next process executed after the priority-setting process places its *sched_setparam*() function call.

The implementation shall document the time-stamp mechanism used for the tests. All timings shall be adjusted for the time required for the time-stamp operations.

The implementor is free to choose the method by which time-stamps taken in the priority-setting and target processes are compared but shall document the method used.

The documentation shall include the source for all tests.

G.10 Clocks and Timers

Because realtime systems are based heavily on the accuracy of the timers in the system, an implementation should provide accurate measurements of its timers. All timer measurements are taken with respect to the highest priority process.

If the implementation supports both the Timers and Realtime Performance Documentation options, it shall report

Clock Drift

This is a measure of the worst-case, long-term (cumulative) deviation of a clock. It is a cyclic deviation of long period that appears to vary linearly with time and is expressed as percent deviation per month. For example, a watch might have a drift of 5 s every month (0.0002%/month).

The drift should be measured over at least 24 h. The measurement period shall be reported.

Clock Jitter

This is a measure of the worst-case, short-term deviation or repeatability of a clock. It is a cyclic deviation of short period and is expressed as percent deviation per second. For example, a clock might have a jitter of 50 μs from one second to the next (0.001%/s), perhaps because it must implement a 60 Hz clock from a higher resolution counter whose rate is not a multiple of 60 Hz.

The measurement period shall be reported if this measurement is documented.

Clock/Timer Granularity

This is a measurement of the resolution of the various clocks and timers. The numbers returned by *clock_getres*() (for *clock_id* = CLOCK_REALTIME) should be published. In addition, if the resolutions supported by the *nanosleep*() function or of timers created with a *clock_id* = CLOCK_REALTIME are different from that returned by *clock_getres*() (for *clock_id* = CLOCK_REALTIME), then these resolutions should also be documented.

Time Setting

The time for a user process to set a clock and have the system return to the caller.

Time Getting

The time for a user process to read a clock and have the system return to the caller.

Timer Arming—Absolute Time, One-Shot

The time for a user process to set and arm a one-shot per-process timer, using *timer_settime*() with an absolute time, and have the system return to the caller.

Timer Arming—Absolute Time, Periodic

The time for a user process to set and arm a periodic per-process timer, using *timer_settime*() with an absolute time, and have the system return to the caller.

Timer Arming—Relative Time, One-Shot

The time for a user process to set and arm a one-shot per-process timer, using *timer_settime*() with a relative time, and have the system return to the caller.

Timer Arming—Relative Time, Periodic

The time for a user process to set and arm a periodic per-process timer, using *timer_settime*() with a relative time, and have the system return to the caller.

Timer Disarming—Absolute Time, One-Shot

The time for a user process to disarm a one-shot per-process timer, using *timer_settime*() with an absolute time, and have the system return to the caller.

Timer Disarming—Absolute Time, Periodic

The time for a user process to disarm a periodic per-process timer, using *timer_settime*() with an absolute time, and have the system return to the caller.

Timer Disarming—Relative Time, One-Shot

The time for a user process to disarm a one-shot per-process timer, using *timer_settime*() with a relative time, and have the system return to the caller.

Timer Disarming—Relative Time, Periodic

The time for a user process to disarm a periodic per-process timer, using *timer_settime*() with a relative time, and have the system return to the caller.

Timer Reload Time

The amount of time required to reload and rearm a timer when a periodic timer expires.

Timer Expiration Service Latency—Absolute, One-Shot

This is the time interval between the timer expiration—that is, the absolute time specified in the call to *timer_settime*()—and the first instruction to be executed in the event handler for a one-shot timer.

In some cases this might be a fixed interval (for example, it is always at least 50 µs) plus some variance. Both the interval and the variance should be reported.

Timer Expiration Service Latency—Absolute, Periodic

This is the time interval between the timer expiration—that is, the absolute time specified in the call to *timer_settime*()—and the first instruction to be executed in the event handler for the first expiration of an absolute, periodic timer. In some cases this might be a fixed interval (for example, it is always at least 50 µs) plus some variance. Both the interval and the variance should be reported.

Timer Expiration Service Latency—Relative, One-Shot

This is the time interval between the timer expiration—that is, the interval specified in the call to *timer_settime*() plus the current clock time at the time of the call—and the first instruction to be executed in the event handler for a one-shot relative timer. In some cases this might be a fixed interval (for example, it is always at least 50 µs) plus some variance. Both the interval and the variance should be reported.

Timer Expiration Service Latency—Relative, Periodic

This is the time interval between the timer expiration—that is, the interval specified in the call to *timer_settime*() plus the current clock time at the time of the call—and the first instruction to be executed in the event handler for a periodic relative timer. In some cases this might be a fixed interval (for example, it is always at least 50 µs) plus some variance. Both the interval and the variance should be reported.

G.11 Message Passing

If the implementation supports both the Message Passing and Realtime Performance Documentation options, it shall report the metrics listed below. The metric report shall include sufficient information that the measurements can be reproduced.

MQ Open Time

This metric is the time for a single successful first *mq_open*() of a message queue.

MQ Close Time

This metric is the time for a single successful last *mq_close*() of a message queue.

MQ Send Times

This metric is the time for a successful *mq_send*() for messages of the following *msg_len*: 0, 4, 7, 16, 32.

MQ Transfer Rates

This metric is the rate (number of bytes of message data divided by time) for successful *mq_send*() to *mq_receive*() for messages of the following *msg_len*: 0, 4, 7, 16, 32. The methodology for measuring the transfer rate is implementation defined.

MQ Receive Rates

This metric is the rate (time divided by number of bytes of message data) for a successful *mq_receive*() from a queue already containing a single message for messages of the following lengths: 0, 4, 7, 16, 32.

G.12 Thread Management

If the implementation supports both the Thread and Realtime Performance Documentation options, it shall report

Granularity of Parallelism: Light Load with *n* Threads

This is the minimum number of iterations of a null loop that have to be executed in *n* threads simultaneously before the time needed by the *n* threads is less than the time needed for a single thread to execute the total number of iterations by itself. The time for the *n* thread case has to include the time to create all *n* threads and to wait for them to terminate. This number is used by a programmer to determine when it might be advantageous to divide a task into *n* different pieces that can be executed simultaneously. This metric should be provided for *n* between one and the number of processors on the machine.

Granularity of Parallelism: Heavy Load Overhead of *x* Percent

On a machine with *p* processors, this is the minimum number of iterations of a null loop such that running *4*p* copies of this loop takes no more than *x* percent longer than running 2**p* copies of the loop with twice that number of iterations. The time needed has to include the time to create all of the threads and to wait for them to terminate. The metric shall be provided for *x* equal 1%, 5%, 10%, and 20%.

Once Overhead

This is the time needed for the highest priority thread to execute the *pthread_once*() function when the *init_routine* has already been executed.

Self Overhead

This is the time needed for the highest priority thread to perform the *pthread_self*() operation for the following numbers of threads: 1, 21, 101, 1023.

G.13 Thread Cancellation

Cancellation Response

This is the time interval between the issuing of a cancellation request to a thread executing an infinite loop with asynchronous cancelability enabled and the start of the execution of its first cancellation cleanup handler.

Handler Registration

This is the time interval needed to register a handler and a non-**NULL** argument with the *pthread_cleanup_push*() function when no other application-defined handlers are already registered.

Handler De-registration

This is the time interval needed to deregister a handler with the *pthread_cleanup_pop*() function when it is the only application-defined handler registered. The handler is not executed.

Opening a Cancellation Point

This is the time interval it takes to call *pthread_testcancel*() in a thread with no pending cancellation requests.

Taking a Cancellation Point

This is the time interval between calling *pthread_testcancel*() in a thread with a pending cancellation request and the start of the execution of its first cancellation cleanup handler.

Annex H Realtime Files

(Informative)

This annex defines an interface that allows an application to specify various characteristics regarding how its normal file requests [such as *read*() and *write*()] should be handled. This is an informative annex, placing no requirements on implementations. It is expected that this annex will be removed in a later amendment when a file advisory interface is made part of the standard.

This facility provides a mechanism for the manipulation of realtime attributes of files in two component parts:

1) *The Interface*: An interface that allows an application to obtain information about and influence system usage of a file system, and
2) *Attributes and Capabilities*: A set of defined types of information that make this interface useful in a rotating storage media model.
   - *Capabilities*: These are parameters that are constant with (direct) respect to the application, such as the transfer granularity, disk geometry based parameters, and so on.
   - *Attributes*: These are parameters that the application may control, such as the number of blocks pre-reserved for a file and what sort of access the application will use (for example, random versus sequential).

The following set of attributes exist:

1) Sequential access
2) Pre-allocation
3) Direct I/O
4) Cache usage
5) Aligned transfers
6) Transfer granularity

Passage of the attributes is accomplished via a structure.

Lastly, the interface contains an ability to obtain a buffer that is placed in memory according to desired constraints.

The state or value of realtime attributes of a file shall be persistent from the time of setting of the attribute until last close of the file.

Conforming implementations are not required to make realtime attributes persistent after last close of the file.

A Strictly Conforming Application shall not depend upon persistence of realtime attributes of a file after last close.

With one exception, conforming implementations are not required to make real-time attributes persistent across different open instances of the same file. That exception is that the *atb_alloc* attribute shall be persistent across all open instances of a file.

A conforming implementation may support file systems to which only a subset of attributes applies.

H.1 Data Definitions for Realtime Files

The header file `<rtfiles.h>` defines the symbols, types, and manifest constants used by the realtime file facility.

Inclusion of the `<rtfiles.h>` header shall make visible the symbols defined by this part of ISO/IEC 9945 to be in the header `<sys/types.h>`.

H.1.1 Realtime Files Specification Structures

H.1.1.1 The Allocation Attribute Buffer Structure

The attribute buffer structure *rf_attrbuf* specifies the attributes of a file and includes the following members:

Member Type	Member Name	Description
off_t	*atb_alloc*	File allocation length
int	*atb_allocflags*	Allocation flags
int	*atb_cacheflags*	File usage pattern
int	*atb_dioflags*	Direct I/O flags
size_t	*atb_diosize*	Direct I/O transfer size

Implementations may add extensions as permitted in 1.3.1.1, item (2). Adding extensions to this structure, which may change the behavior of the application with respect to this standard when those fields in the structure are uninitialized, also requires that the extension be enabled as required by 1.3.1.1.

The *atb_alloc* field is the length in bytes of the space to be allocated for future use as part of that file. The allocation has the following characteristics:

> The allocation amount is independent of file size as returned by the *stat*() function, and is reduced to zero when the file is truncated.

The *atb_allocflags* field shall have a value formed from the inclusive OR of zero or more of the following flags defined in `<rtfiles.h>` that pertain to the type of allocation required:

ATB_SEQ

> Advisory indicating that allocation shall be done in a manner consistent with sequential access. If such allocation is not possible, *rf_create*() and *rf_setattr*() shall fail. When returned in *actattr*, this flag indicates that the file was made sequential. If both ATB_ADVSEQ and ATB_SEQ are specified, ATBSEQ takes precedence and ATB_ADVSEQ is ignored.

ATB_ADVSEQ

> Advisory indicating that the file is to be primarily accessed sequentially; that is, that the system should optimize placement for sequential access if possible.

The *atb_cacheflags* field shall have a value formed from the inclusive OR of zero or more of the following flags defined in `<rtfiles.h>` that pertain to the type of caching desired:

ATB_CACHENOREUSE

> Advisory indicating that recently accessed data blocks are not likely to be accessed again in the near term.

ATB_CACHERANDOM

Advisory indicating that the application will be accessing the file randomly.

ATB_CACHESEQUENTIAL

Advisory indicating that the application will be accessing the file sequentially.

The *atb_dioflags* field shall have a value formed from the inclusive OR of zero or more of the following flags defined in `<rtfiles.h>` that pertain to direct I/O:

ATB_DIOENABLE

Advisory indicating that the file can be enabled for direct I/O. When this attribute is set for an open file description, subsequent *read*() and *write*() operations shall perform direct I/O.

The *atb_diosize* field contains the direct I/O transfer size in bytes for files that require a single direct I/O transfer size to be defined. This field shall be returned as zero unless the underlying file system restricts direct I/O to a single transfer size.

The flags ATB_CACHENOREUSE, ATB_CACHERANDOM, ATB_CACHESEQUENTIAL, and ATB_-DIOENABLE shall be associated with the open file description.

H.1.1.2 The Allocation Capabilities Buffer

The allocation capabilities buffer structure *rf_capallocbuf* specifies the allocation capabilities of a file or file system and includes the following members:

Member Type	**Member Name**	**Description**
off_t	*atc_allocmin*	Minimum preallocation size
off_t	*atc_allocmax*	Maximum preallocation size
off_t	*atc_allocincr*	Granularity of file space allocation
int	*atc_allocincrtype*	
int	*atc_allocflags*	Allocation flags

Implementations may add extensions as permitted in 1.3.1.1, item (2). Adding extensions to this structure, which may change the behavior of the application with respect to this standard when those fields in the structure are uninitialized, also requires that the extension be enabled as required by 1.3.1.1.

The members of the structure are defined as follows:

The *atc_allocmin* field contains the minimum preallocation size in bytes.
The *atc_allocmax* field contains the maximum preallocation size in bytes.
This field shall be returned as zero if preallocation is not supported.
The *atc_allocincr* field contains the granularity of increment for file space allocation.
The following values of *atc_allocincrtype* defined in `<rtfiles.h>` indicate how to interpret the granularity of increment.

ATC_MULTIPLY

Allocation size shall be an integral multiple of *atc_allocincr*.

ATC_EXPONENT

Allocation size shall be an integral power of *atc_allocincr*.

ATC_LIST

Possible allocation values are a list with *atc_allocincr* entries.

If the bitwise AND of *atc_allocflags* and any of the following flags defined in `<rtfiles.h>` is nonzero, the associated preallocation attribute is provided:

ATC_SEQ

Sequential allocation is possible.

H.1.1.3 The Cache Capabilities Buffer

The cache capabilities buffer structure *rf_capcachebuf* specifies the cache capabilities of a file or file system and includes the following members:

Member Type	Member Name	Description
int	*atc_cacheflags*	Supported caching options

Implementations may add extensions as permitted in 1.3.1.1, item (2). Adding extensions to this structure, which may change the behavior of the application with respect to this standard when those fields in the structure are uninitialized, also requires that the extension be enabled as required by 1.3.1.1.

The members of the structure are defined as follows:

The *atc_cacheflags* field indicates which of the following caching options are supported:

ATC_CACHENOREUSE

The implementation provides ATB_CACHENOREUSE functionality.

ATC_CACHERANDOM

The implementation provides ATB_CACHERANDOM functionality.

ATC_CACHESEQUENTIAL

The implementation provides ATB_CACHESEQUENTIAL functionality.

H.1.1.4 The Buffered I/O Capabilities Buffer

The buffered I/O capabilities buffer structure *rf_capbiobuf* specifies the buffered I/O capabilities of a file or file system and includes the following members:

Member Type	Member Name	Description
off_t	*atc_boffset*	Optimal seek alignment
size_t	*atc_biomin*	Minimum recommended transfer size
size_t	*atc_biomax*	Maximum recommended transfer size
size_t	*atc_bioincr*	Transfer size increment
int	*atc_bioincrtype*	

Implementations may add extensions as permitted in 1.3.1.1, item (2). Adding extensions to this structure, which may change the behavior of the application with respect to this standard when those fields in the structure are uninitialized, also requires that the extension be enabled as required by 1.3.1.1.

The members of the structure are defined as follows:

The *atc_boffset* field indicates the seek alignment in bytes advised for best performance.
The *atc_biomin* field indicates the minimum transfer size advised for best performance.
The *atc_biomax* field indicates the maximum transfer size advised for best performance.
The *atc_bioincr* field indicates the transfer size increment (see H.1.1.2).
The *atc_bioincrtype* field indicates how to interpret *atc_bioincr* (see H.1.1.2).

H.1.1.5 The Atomic I/O Capabilities Buffer

The atomic I/O capabilities buffer structure *rf_capaiobuf* specifies the atomic I/O capabilities of a file or file system and includes the following members:

Member Type	Member Name	Description
size_t	*atc_aiomin*	Minimum atomic transfer size
size_t	*atc_aiomax*	Maximum atomic transfer size
size_t	*atc_aioincr*	Atomic transfer size increment
int	*atc_aioincrtype*	
off_t	*atc_aiosoffset*	Atomic seek alignment required
off_t	*atc_aioboundary*	

Implementations may add extensions as permitted in 1.3.1.1, item (2). Adding extensions to this structure, which may change the behavior of the application with respect to this standard when those fields in the structure are uninitialized, also requires that the extension be enabled as required by 1.3.1.1.

The members of the structure are defined as follows:

The *atc_aiomin* field indicates the minimum atomic transfer size supported. An atomic I/O operation is a transfer of data that succeeds or fails as a unit and that is not divisible by another transfer to the same addresses in the data storage involved.
The *atc_aiomax* field indicates the maximum atomic transfer size supported.
The *atc_aioincr* field indicates the transfer size increment (see H.1.1.2).
The *atc_aioincrtype* field indicates how to interpret atc_aioincr (see H.1.1.2).
The *atc_aiosoffset* indicates starting seek alignment required for atomic transfer, if any.
The *atc_aioboundary* indicates a seek alignment that cannot be spanned by an atomic transfer, if any.

H.1.1.6 The Direct I/O Capabilities Buffer

The direct I/O capabilities buffer structure *rf_capdiobuf* specifies the direct I/O capabilities of a file or file system and includes the following members:

Member Type	Member Name	Description
off_t	*atc_doffset*	Direct I/O seek alignment required
size_t	*atc_dalign*	Direct I/O memory alignment required
size_t	*atc_diomin*	Minimum direct I/O transfer size
size_t	*atc_diomax*	Maximum direct I/O transfer size
size_t	*atc_dioincr*	Direct I/O transfer size increment
int	*atc_dioincrtype*	
int	*atc_dioflags*	Direct I/O flags

Implementations may add extensions as permitted in 1.3.1.1, item (2). Adding extensions to this structure, which may change the behavior of the application with respect to this standard when those fields in the structure are uninitialized, also requires that the extension be enabled as required by 1.3.1.1.

The members of the structure are defined as follows:

The *atc_doffset* field indicates a seek alignment required for direct I/O.
The *atc_dalign* field indicates the memory alignment required for direct I/O. A buffer used for direct I/O shall start at an address that is a multiple of this value. A direct read or write operation using a buffer that is not properly aligned may fail, with *errno* set to [EFAULT].
The *atc_diomin* field indicates the minimum transfer size for direct I/O.
The *atc_diomax* field indicates the maximum transfer size for direct I/O.
The *atc_dioincr* field indicates the transfer size increment (see H.1.1.2).
The *atc_dioincrtype* field indicates how to interpret *atc_dioincr* (see H.1.1.2).
If the bitwise AND of *atc_dioflags* and any of the following flags defined in `<rtfiles.h>` is nonzero, the associated direct I/O attribute is provided:

ATC_DIOCAPABLE

Direct I/O may be performed on this file.

ATC_DIOREQUIRED

Direct I/O shall be performed on this file.

For any capability attributes that are not applicable to the associated file or file system, zero shall be returned as the attribute value.

H.1.1.7 Which Argument Formats

A *reqwhich* argument shall have a value formed from the inclusive OR of zero or 243 more of the following flags defined in `<rtfiles.h>`, each indicating the associated attributes:

ATB_ALLOC

Preallocation size.

ATB_ALLOCFLAGS

Preallocation flags.

ATB_CACHEFLAGS

Caching flags.

ATB_DIOFLAGS

Direct I/O flags.

ATB_DIOSIZE

Direct I/O size.

ATB_ALL

All of the above.

A nonzero value for the bitwise AND of the updated value of the *actwhich* argument, returned by the *rf_create*() and *rf_setattr*() functions, and any of the values defined for the *reqwhich* argument indicates that the associated requested attributes buffer does not match the corresponding element in the associated actual attributes buffer. A zero value in *actwhich* indicates that all requested attributes have been set as requested.

H.2 Realtime File Functions

H.2.1 Create a Realtime File

Function: *rf_create*()

H.2.1.1 Synopsis

```
#include <rtfiles.h>
int rf_create(const char *path, mode_t mode,
            const struct rf_attrbuf *reqattr, int reqwhich,
            struct rf_attrbuf *actattr, int *actwhich);
```

H.2.1.2 Description

If {_POSIX_REALTIME_FILES} is defined:

The *rf_create*() function creates a file with realtime attributes. The *path* and *mode* arguments are the same as for *creat*()(see 5.3.2).
The *reqattr* and *actattr* arguments point to structures of type *rf_attrbuf*.
The requested attributes of the created file are specified the *rf_attrbuf* structure specified by the *reqattr* argument.

The actual attributes of the created file shall be stored in the *rf_attrbuf* structure specified by the *actattr* argument upon successful completion of the *rf_create*() function.
The *reqwhich* argument shall be as described for the *reqwhich* argument in H.1.1.7.
The *actwhich* argument shall be as described for the *actwhich* argument in H.1.1.7.

Otherwise:

Either the implementation shall support the *rf_create*() function as described above or the *rf_create*() function shall fail.

H.2.1.3 Returns

Upon successful completion, the function shall open the file and return a nonnegative integer representing the lowest numbered unused file descriptor. Otherwise it shall return –1 and set *errno* to indicate the error.

H.2.1.4 Errors

If any of the following conditions occur, the *rf_create*() function shall return –1 and set *errno* to the corresponding value:

[EACCES]	Search permission is denied on a component of the path prefix, or the file exists and write permission is denied, or the file does not exist and write permission is denied for the parent directory of the file to be created.
[EEXIST]	The specified file already exists.
[EINTR]	The *rf_create*() operation was interrupted by a signal.
[EINVAL]	The *rf_create*() function is not supported for the specified file.
	One or more of the requested attributes are not valid. Upon return, if one or more of the attributes was not valid, *actwhich* shall indicate at least one of the parameters found to be invalid.
[EMFILE]	Too many file descriptors are currently in use by this process.
[ENAMETOOLONG]	
	The length of the *path* string exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.
[ENFILE]	Too many files are currently open in the system.
[ENOENT]	Either the path prefix does not exist or the *path* argument points to an empty string.
[ENOSPC]	The directory or file system that would contain the new file cannot be extended.
[ENOSYS]	The function *rf_create* is not supported by this implementation.
[ENOTDIR]	A component of the path prefix is not a directory.
[EPERM]	The process does not have the appropriate privilege.
[EROFS]	The requested operation requires writing in a directory on a read-only file system.

H.2.1.5 Cross-References

creat(), 5.3.2; *open*(), 5.3.1.

H.2.2 Get Attributes of Realtime File

Function: *rf_getattr*()

H.2.2.1 Synopsis

```
#include <rtfiles.h>
int rf_getattr(int fildes, struct rf_attrbuf *actattr);
```

H.2.2.2 Description

If {_POSIX_REALTIME_FILES} is defined:

> The *rf_getattr*() function allows the calling process to obtain the realtime attributes of the file specified by *fildes*. The fields of the *rf_attrbuf* structure referenced by *actattr* need not match those used when the file was created or the states of the attributes after use of the *rf_setattr*() function. Implementations are not required to make all attributes persistent across open file descriptions of a file.

Otherwise:

> Either the implementation shall support the *rf_getattr*() function as described above or the *rf_getattr*() function shall fail.

H.2.2.3 Returns

The *rf_getattr*() function shall return zero if the function is successful; otherwise, the function shall return –1 and set *errno* to indicate the error.

H.2.2.4 Errors

If any of the following conditions occur, the *rf_getattr*() function shall return –1 and set *errno* to the corresponding value:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINVAL]	The *rf_getattr*() function is not supported for the specified file.
[ENOSYS]	The function *rf_getattr*()is not supported by this implementation.

H.2.2.5 Cross-References

rf_create(), H.2.1; *rf_setattr*(), H.2.3.

H.2.3 Set Attributes of Realtime File

Function: *rf_setattr*()

H.2.3.1 Synopsis

```
#include <rtfiles.h>
int rf_setattr(int fildes, const struct rf_attrbuf *reqattr,
            int reqwhich, struct rf_attrbuf *actattr, int *actwhich);
```

H.2.3.2 Description

If {_POSIX_REALTIME_FILES} is defined:

> The *rf_setattr*() function sets one or more realtime attributes associated with the file specified by *fildes*. The *reqwhich* argument shall be as described for the *reqwhich* argument in H.1.1.7.

The *reqattr* and *actattr* arguments point to structures of type *rf_attrbuf*.
The requested attributes of the file are specified in the *rf_attrbuf* structure specified by the *reqattr* argument.
The resulting attributes for the file shall be stored in *actattr* upon successful completion of the function.
The *actwhich* argument shall be as described for the *actwhich* argument in H.1.1.7.

Otherwise:

Either the implementation shall support the *rf_setattr*() function as described above or the *rf_setattr*() function shall fail.

H.2.3.3 Returns

The *rf_setattr*()function shall return zero if the new attributes were successfully 375 set. The function shall return −1 if at least one of the requested operation(s) cannot be performed. If multiple attributes were specified, and one or more attributes could not be set, the resultant values in *actattr* are undefined. Implementations may note one, some, or all of the attributes in error. A Strictly Conforming Application shall not rely on a particular behavior in this regard.

H.2.3.4 Errors

If any of the following conditions occur, the *rf_setattr*() function shall return −1 and set *errno* to the corresponding value:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINVAL]	The *rf_setattr*() function is not supported for the specified file.
	One of the requested attributes was not valid or is in conflict with an attribute specified on another open instance of this file. Upon return, *actwhich* shall indicate which parameter was found to be in error. No attributes of the file are changed.
[ENOSYS]	The function *rf_setattr*() is not supported by this implementation.

H.2.3.5 Cross-References

rf_create(), H.2.1; *rf_getattr*(), H.2.2.

H.2.4 Get Allocation Capabilities of Realtime Files and File Systems

Function: *rf_getalloccap*()

H.2.4.1 Synopsis

```
#include <rtfiles.h>
int rf_getalloccap(int fildes, struct rf_capallocbuf *capbufp);
```

H.2.4.2 Description

If {_POSIX_REALTIME_FILES} is defined:

The *rf_getalloccap*() function allows the calling process to obtain the allocation capabilities of the file specified by *fildes*.
If *fildes* specifies a file that is not a directory, then the capabilities refer to the file itself. If *fildes* specifies a file that is a directory, then the capabilities refer to files created within that directory.
The *capbufp* argument points to a structure of type *rf_capallocbuf*.

The capabilities of the file or file system shall be stored in the *rf_capallocbuf* structure specified by the *capbufp* argument upon successful completion of the *rf_getalloccap*() function.

Otherwise:

Either the implementation shall support the *rf_getalloccap*() function as described above or the *rf_getalloccap*() function shall fail.

H.2.4.3 Returns

The *rf_getalloccap*() function shall return zero if the function is successful; otherwise, the function shall return –1 and set *errno* to indicate the error.

H.2.4.4 Errors

If any of the following conditions occur, the *rf_getalloccap*() function shall return –1 and set *errno* to the corresponding value:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINVAL]	The *rf_getalloccap*() function is not supported for the specified file.
[ENOSYS]	The function *rf_getalloccap*() is not supported by this implementation.

H.2.5 Get Cache Capabilities of Realtime Files and File Systems

Function: *rf_getcachecap*()

H.2.5.1 Synopsis

```
#include <rtfiles.h>
int rf_getcachecap(int fildes, struct rf_capcachebuf *capbufp);
```

H.2.5.2 Description

If {_POSIX_REALTIME_FILES} is defined:

The *rf_getcachecap*() function allows the calling process to obtain the cache capabilities of the file specified by *fildes*.
If *fildes* specifies a file that is not a directory, then the capabilities refer to the file itself. If *fildes* specifies a file that is a directory, then the capabilities refer to files created within that directory.
The capbufp argument points to a structure of type *rf_capcachebuf*.
The capabilities of the file or file system shall be stored in the *rf_capcachebuf* structure specified by the *capbufp* argument upon successful completion of the *rf_getcachecap*() function.

Otherwise:

Either the implementation shall support the *rf_getcachecap*() function as described above or the *rf_getcachecap*() function shall fail.

H.2.5.3 Returns

The *rf_getcachecap*() function shall return zero if the function is successful; otherwise, the function shall return –1 and set *errno* to indicate the error.

H.2.5.4 Errors

If any of the following conditions occur, the *rf_getcachecap*() function shall return –1 and set *errno* to the corresponding value:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINVAL]	The *rf_getcachecap*() function is not supported for the specified file.
[ENOSYS]	The function *rf_getcachecap*() is not supported by this implementation.

H.2.6 Get Buffered I/O Capabilities of Realtime Files and File Systems

Function: *rf_getbiocap*()

H.2.6.1 Synopsis

```
#include <rtfiles.h>
int rf_getbiocap(int fildes, struct rf_capbiobuf *capbufp);
```

H.2.6.2 Description

If {_POSIX_REALTIME_FILES} is defined:

> The *rf_getbiocap*() function allows the calling process to obtain the buffered I/O capabilities of the file specified by *fildes*.
> If *fildes* specifies a file that is not a directory, then the capabilities refer to the file itself. If *fildes* specifies a file that is a directory, then the capabilities refer to files created within that directory.
> The *capbufp* argument points to a structure of type *rf_capbiobuf*.
> The capabilities of the file or file system shall be stored in the *rf_capbiobuf* structure specified by the *capbufp* argument upon successful completion of the *rf_getbiocap*() function.

Otherwise:

> Either the implementation shall support the *rf_getbiocap*() function as described above or the *rf_getbiocap*() function shall fail.

H.2.6.3 Returns

The *rf_getbiocap*() function shall return zero if the function is successful; otherwise, the function shall return –1 and set *errno* to indicate the error.

H.2.6.4 Errors

If any of the following conditions occur, the *rf_getbiocap*() function shall return –1 and set *errno* to the corresponding value:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINVAL]	The *rf_getbiocap*() function is not supported for the specified file.
[ENOSYS]	The function *rf_getbiocap*() is not supported by this implementation.

H.2.7 Get Atomic I/O Capabilities of Realtime Files and File Systems

Function: *rf_getaiocap*()

H.2.7.1 Synopsis

```
#include <rtfiles.h>
int rf_getaiocap(int fildes, struct rf_capaiobuf *capbufp);
```

H.2.7.2 Description

If {_POSIX_REALTIME_FILES} is defined:

> The *rf_getaiocap*() function allows the calling process to obtain the atomic I/O capabilities of the file specified by *fildes*.
> If *fildes* specifies a file that is not a directory, then the capabilities refer to the file itself. If *fildes* specifies a file that is a directory, then the capabilities refer to files created within that directory.
> The *capbufp* argument points to a structure of type *rf_capaiobuf*.
> The capabilities of the file or file system shall be stored in the *rf_capaiobuf* structure specified by the *capbufp* argument upon successful completion of the *rf_getaiocap*() function.

Otherwise:

> Either the implementation shall support the *rf_getaiocap*() function as described above or the *rf_getaiocap*() function shall fail.

H.2.7.3 Returns

The *rf_getaiocap*() function shall return zero if the function is successful; otherwise, the function shall return –1 and set *errno* to indicate the error.

H.2.7.4 Errors

If any of the following conditions occur, the *rf_getaiocap*() function shall return –1 and set *errno* to the corresponding value:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINVAL]	The *rf_getaiocap*() function is not supported for the specified file.
[ENOSYS]	The function *rf_getaiocap*() is not supported by this implementation.

H.2.8 Get Direct I/O Capabilities of Realtime Files and File Systems

Function: *rf_getdiocap*()

H.2.8.1 Synopsis

```
#include <rtfiles.h>
int rf_getdiocap(int fildes, struct rf_capdiobuf *capbufp);
```

H.2.8.2 Description

If {_POSIX_REALTIME_FILES} is defined:

> The *rf_getdiocap*() function allows the calling process to obtain the direct I/O capabilities of the file specified by *fildes*.
> If *fildes* specifies a file that is not a directory, then the capabilities refer to the file itself. If *fildes* specifies a file that is a directory, then the capabilities refer to files created within that directory.

The *capbufp* argument points to a structure of type *rf_capdiobuf*.
The capabilities of the file or file system shall be stored in the *rf_capdiobuf* structure specified by the *capbufp* argument upon successful completion of the *rf_getdiocap*() function.

Otherwise:

Either the implementation shall support the *rf_getdiocap*() function as described above or the *rf_getdiocap*() function shall fail.

H.2.8.3 Returns

The *rf_getdiocap*() function shall return zero if the function is successful; otherwise, the function shall return −1 and set *errno* to indicate the error.

H.2.8.4 Errors

If any of the following conditions occur, the *rf_getdiocap*() function shall return −1 and set *errno* to the corresponding value:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINVAL]	The *rf_getdiocap*() function is not supported for the specified file.
[ENOSYS]	The function *rf_getdiocap*() is not supported by this implementation.

H.2.9 Get Increment Lists

Functions: *rf_getallocincr*(), *rf_getincr*()

H.2.9.1 Synopsis

```
#include <rtfiles.h>
int rf_getallocincr(int fildes, off_t *offbuffer, int buflen);
int rf_getincr(int fildes, int incr, size_t *sizebuffer, int buflen);
```

H.2.9.2 Description

If {_POSIX_REALTIME_FILES} is defined:

The *rf_getallocincr*() function allows the calling process to obtain the allocation increment list of the file specified by *fildes*. The *rf_getincr*() function allows the calling process to obtain the transfer size increment list for a capability attribute of the file specified by *fildes*. If *fildes* specifies a file that is not a directory, then the increment list refers to the file itself. If *fildes* specifies a file that is a directory, then the increment list refers to files created within that directory.
Each value in the increment list is a valid value for the attribute requested. For example, requesting the ATC_BIOINCR attribute increment list for rotating magnetic media may return the sector, track, and cylinder sizes as a three element list.
For the *rf_getincr*() function, the value of the *incr* argument may be any of the following defined in `<rtfiles.h>`, and it specifies which increment list is required:

ATC_AIOINCR

Atomic transfer size increment.

ATC_BIOINCR

Suggested transfer size increment.

ATC_DIOINCR

Direct I/O transfer size increment.

The *offbuffer* argument points to an array of type *off_t* that shall receive the list of allocation increments. The *sizebuffer* argument points to an array of type *size_t* that shall receive the list of transfer size increments. The *buflen* argument indicates the maximum number of list entries that may be returned.

Otherwise:

Either the implementation shall support the *rf_getallocincr*() and *rf_getincr*() functions as described above or each of the *rf_getallocincr*() and *rf_getincr*() functions shall fail.

H.2.9.3 Returns

If successful, the *rf_getallocincr*() function shall return the number of list entries stored in *offbuffer*; otherwise, the function shall return –1 and set *errno* to indicate the error.

If successful, the *rf_getincr*() function shall return the number of list entries stored in *sizebuffer*; otherwise, the function shall return –1 and set *errno* to indicate the error.

H.2.9.4 Errors

If any of the following conditions occur, the *rf_getallocincr*() and *rf_getincr*() functions shall return –1 and set *errno* to the corresponding value:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINVAL]	The function is not supported for the specified file.
	The specified capability does not have an increment list.
	The *incr* argument specifies an invalid increment list (*rf_getincr*() only).
[ENOSYS]	The function is not supported by this implementation.

H.2.10 Allocate a Suitably Aligned Data Buffer

Function: *rf_getbuf*()

H.2.10.1 Synopsis

```
#include <sys/types.h>
void *rf_getbuf(off_t bufsize, off_t align);
```

H.2.10.2 Description

If {_POSIX_REALTIME_FILES} is defined:

The *rf_getbuf*() function returns a pointer to a suitably aligned buffer of size *bufsize*. The buffer shall start on an address that is aligned modulo *align*. The *align* argument is typically taken from the *atc_dalign* element of a *rf_capdiobuf* structure.

Otherwise:

Either the implementation shall support the *rf_getbuf*() function as described above or the *rf_getbuf*() function shall fail.

H.2.10.3 Returns

If successful, the *rf_getbuf*() function shall return a pointer to a suitably aligned buffer; otherwise, the function shall return –1 and set *errno* to indicate the error.

H.2.10.4 Errors

If any of the following conditions occur, the *rf_getbuf*() function shall return –1 and set *errno* to the corresponding value:

[ENOMEM]	A buffer area of the requested size could not be allocated.
[ENOSYS]	The function *rf_getbuf*() is not supported by this implementation.

H.2.10.5 Cross-References

rf_create(), H.2.1; *rf_freebuf*(), H.2.11; *rf_getattr*(), H.2.2; *rf_setattr*(), H.2.3.

H.2.11 Release a Previously Allocated Data Buffer

Function: *rf_freebuf*()

H.2.11.1 Synopsis

```
#include <sys/types.h>
void *rf_freebuf(void *buf);
```

H.2.11.2 Description

If {_POSIX_REALTIME_FILES} is defined:

> The *rf_freebuf*() function releases the buffer referenced by *buf*, which was previously allocated by the *rf_getbuf*() function.

Otherwise:

> Either the implementation shall support the *rf_freebuf*() function as described above or the *rf_freebuf*() function shall fail.

H.2.11.3 Returns

If successful, the *rf_freebuf*() function shall return zero; otherwise, the function shall return –1 and set *errno* to indicate the error.

H.2.11.4 Errors

If any of the following conditions occur, the *rf_freebuf*() function shall return –1 and set *errno* to the corresponding value:

[EINVAL]	The argument *buf* does not point to a buffer allocated by the *rf_getbuf*() function.
[ENOSYS]	The function *rf_freebuf*() is not supported by this implementation.

H.2.11.5 Cross-References

rf_create(), H.2.1; *rf_getattr*(), H.2.2; *rf_getbuf*(), H.2.10; *rf_setattr*(), H.2.3.

H.3 Realtime Files

The purpose of these metrics is twofold:

1) To allow the user to compare the transfer rates of equivalent hardware platforms, and
2) On a given platform, to distinguish performance differences induced by changes in the attributes of files.

NOTE — What is presented here is an overall set of metrics for the total realtime file model presented in this annex. Individual vendors shall interpret this section with regard to each different hardware platform. This interpretation should be consistent with the purpose.

H.3.1 Conformance

If the implementation supports both the Realtime Files and Realtime Performance Documentation options, it shall report, for at least one hardware platform specified by the vendor, all metrics described in this section. A conforming implementation may specify any given metric as "Not Applicable" if the system under test does not support the function tested or if use of that function has no visible effect on file I/O performance.

H.3.2 Fundamental Transfer Size

The first metric defines the context of further metrics.

Fundamental Transfer Size

The vendor shall first specify a number, which, for rotating magnetic media, should correspond to the block size for the device. If this value is other than the block size, the vendor shall document the significance of the value chosen.

H.3.3 Transfer Metrics

For each of the metrics defined here, the vendor shall specify multiple transfer timings. These timings shall be taken under the conditions defined for each metric. The timings specify the time to read and to write a file under each of five different pairs of file size and transfer size. These pairs are

File Size (in bytes)	Transfer Size
65 536	Fundamental transfer size / 2
65 536	Fundamental transfer size
65 536	Optimal transfer size
2 097 152	Fundamental transfer size
2 097 152	Optimal transfer size

If the values 65 536 and 2 097 152 are not integral multiples of the *Fundamental Transfer Size*, the vendor may select the integral multiples of the *Fundamental Transfer Size* that are immediately above these values and shall specify the values used.

If the largest possible file size supported by the implementation is smaller than 65 536 B, the vendor shall provide and document metrics for the largest possible size and approximately 1/32 of that size. If the largest possible file size supported by the implementation is smaller than 2 097 152 B but larger than 65 536 B, the vendor shall provide and document metrics for the largest possible size, for approximately 1/32 of that size, and for 65 536 B. The vendor shall specify the size used.

Basic Metric

No optimizations should be made to the file for this test, such as placing the file in such a way as to enhance performance (i.e., on a cylinder boundary for rotating media). If possible, such optimizations should be actively defeated for this test. The vendor shall document the conditions under which the Basic Metric was derived.

Preallocated File Metric

This metric is achieved by the same tests as the Basic Metric, but with file preallocation and the *atb_cacheflags* having the value ATB_CACHENOREUSE specified when the file is created.

Contiguous File Metric

This metric is achieved by the same tests as the Basic Metric, but with file preallocation and the *atb_cacheflags* having the value ATB_CACHESEQUENTIAL specified when the file is created. *Fundamental Transfer Size*, should be used for the test.

Direct I/O Metric

This metric is derived by the same tests as the Basic Metric, but with direct I/O enabled.

Cache Control Metric

This metric is derived by the same tests as the Basic Metric, but with the cache control attributes set to a value that is optimal for this test. The vendor shall specify what that value is.

Aligned Buffer Metric

This metric is derived by the same tests as the Basic Metric, but with optimal buffer alignment for the test. The vendor shall specify the buffer alignment used for the test.

Best Case Metric

This metric is derived by the same tests as the Basic Metric, but with all attributes of the file optimized for best performance. The vendor shall specify what attribute values were used for this test.

H.4 Rationale for Realtime Files

This subclause defines an interface that allows an application to specify various characteristics regarding how its normal file requests [such as *read*() and *write*()] should be handled.

This facility provides a mechanism for the manipulation of realtime attributes of files in two component parts:

1) *The Interface*: An interface for allowing an application to obtain information about and influence system usage of a file system, and
2) *Attributes and Capabilities:* A set of defined types of information that make this interface useful in a rotating storage media model
   - *Capabilities*: These are parameters that are constant with (direct) respect to the application, such as the transfer granularity, disk geometry based parameters, and so on.

- *Attributes*: These are parameters that the application may control, such as the number of blocks pre-reserved for a file, what sort of access the application will use (for example, random versus sequential).

The following set of attributes exist:

1) Sequential access
2) Pre-allocation
3) Direct I/O
4) Cache Usage
5) Aligned Transfers
6) Transfer Granularity

Passage of the attributes is accomplished via a structure.

Lastly, the interface contains an ability to obtain a buffer that is placed in memory according to desired constraints. This facility does not allocate a buffer, it simply places a buffer within a larger buffer.

The application model used is one that allows the transfer of advisory information between an application and the system. This information can be either system-generated information about the capabilities of the file systems (and other factors in performance, such as memory alignment) available on that system, and application-generated information about its use of the file system. The intent is that the application can vary its behavior in light of the capabilities of the underlying implementation, and that the system may use application "hints" to perform some level of optimization.

This notion is embodied directly in the interface by providing two primary functions for information interchange: The capabilities buffer and its associated functions, and the attributes buffer and its associated functions.

In addition to this information exchange interface, those individual capabilities and attributes that are almost universally provided are also defined in the interface to facilitate standardized usage of common practice.

It must be noted at this point that the standardization of the interface in no way implies that the implementation must implement any particular optimization. It must simply disclaim existence of the optimization in a standard way. Further, the names used for the facilities need not imply any implementation. For example, rather than using the term "contiguous," the abstraction "optimized for sequential access" is used, since this is indeed the notion that the application wishes to express.

The following performance enhancement notions are embodied in the interface:

- *"Contiguous" files*. The notion here is that the access to these files should be optimized for sequential access, probably with overlapped and/or large buffered operations. This is expressed as a sequential advisory in the standard and does *not* imply that the underlying implementation be contiguous.
- *Pre-allocation*.The notion here is twofold. First is the pre-reservation of space to guarantee success while writing a given amount of data. Second is the notion of actual pre-allocation, to remove the latency of allocation from individual writes during data gathering. Once again, the implementation may provide none, one, or both of these actual capabilities.
- *Extension*. The notion here is that when a file is extended, the application may desire that a size other than the fundamental transfer size be pre-reserved or pre-allocated to enhance performance.
- *Direct I/O*. The notion here is to defeat some level of system buffering to increase overall throughput of data to the file system. Doing so may constrain alignment of buffers, etc.
- *Cache usage*. The notion here is that data caching systems earn their performance increase statistically by providing more advantage on some I/O than they lose on others. By providing hints to the system about usage, the application can help the system to optimize cache usage.

— *Aligned transfers*. As mentioned above, direct I/O may require that some fundamental alignment be used, and "normal" I/O may benefit from such alignment. The notion here is to provide information about such transfers.
— *Transfer granularity*. Transfers may also be optimized by using a transfer size that is particularly friendly to the underlying implementation. This notion allows for the expression of transfer sizes.

Additionally, several attributes of files that are not directly related to performance were targeted for standardization. Although the task of standardizing an interface to myriad ways of achieving adequate performance for realtime file usage is difficult, some of the techniques that are common practice are used widely enough, at least in their abstraction, that they may be represented via common interfaces. Such interfaces allow some notions to be specified in a transparently portable way and enhance the portability of less common techniques by defining a standard interface. This can be achieved by defining an interface between the application and the system that is abstract; that is, that defines a protocol for the passage of information regarding devices, file structures, files, and the usage of the file by the application without defining the nature of the data to be transported. The interface also defines the set of data that can be used to describe a very common model of higher performance files, a model of contiguous files on rotating storage media, and he use of "direct" I/O to enhance transfer rate.

Some areas that would suggest themselves for inclusion in his standard are not included for a variety of reasons:

— *Guaranteed delivery*. The insured delivery of data to the underlying storage media, as opposed to system buffers, is considered beyond the scope of this interface, and is addressed in 6.6.
— *Bounded performance*. Consideration was given to a model where the application merely characterized the performance that it needed and the system took care of adjusting file attributes accordingly; it was noted that this is not common practice. This lack of common practice was in some small way related to the fact that the proposed interface might not be implementable. In any case, no consensus could be achieved on some of the interface details, and it was deemed unlikely that a standard could be defined until the state of the art in this area is improved.
— *Circular file*. Consideration was given to files of a fixed size whose contents were written in a circular fashion. (For example, after reaching the size limit of the file, subsequent *write*() functions would overwrite the beginning of the file.) This was requested primarily for logging operations. It was felt that this facility could be implemented as a library function using functions already defined and therefore need not be addressed in this part of ISO/IEC 9945.

With the exception of pre-allocation (*atb_alloc*), persistence of file attributes applies only on an open instance basis. More persistent implementations are allowed but not required. Since pre-allocation of a file pertains to future use of the file, it is an attribute of the file. Consequently, pre-allocation is persistent across all open instances of the file.

H.5 Realtime File Functions

The *stat*() and *fcntl*() functions were not extended since the additions necessary would exceed the current size of those interfaces, and some of the facilities required (such as increment lists, capabilities, and the function of the *which* argument) could not be expressed in that interface. Additionally, there is an explicit effort to provide interfaces that promote type checking. The *fcntl*() function already violates this practice, and it was deemed undesirable to extend its use.

In order to make the interface extensible and to reduce the likelihood of overlapping identifiers with the names of existing or new attributes or capabilities, the interface uses constant names that are prefixed with ATC_ or ATB_ and uses structure members that are prefixed with either *atc_* or *atb_*. Applications should avoid using any identifiers of that form, so that future use of the Realtime Files interface in programs that do not currently use it and future extensions to the attributes and or capabilities will not cause name conflicts.

H.5.1 Data Definitions for Realtime Files

It is not immediately obvious why there are different attributes for sequential access allocation and cache advisory. These exist because one is an attribute of the way the file is placed (traditionally implying contiguous allocation), and one is an attribute of an open instance (typically enabling or disabling read-ahead cache behavior.)

H.5.2 Realtime Files Specification Structures

Due to the complexity of a single large *capbuf* structure, this structure is separated into the four different capabilities structures, and the functions separated.

H.5.2.1 The Allocation Attribute Buffer

There is no specific rationale for this subclause.

H.5.2.2 The Allocation Capabilities Buffer

There is no specific rationale for this subclause.

H.5.2.3 The Cache Capabilities Buffer

There is no specific rationale for this subclause.

H.5.2.4 The Buffered I/O Capabilities Buffer

There is no specific rationale for this subclause.

H.5.2.5 The Atomic I/O Capabilities Buffer

Atomic I/O pertains to both buffered and direct I/O. The conditions for atomic I/O may be different for buffered and direct I/O. The *rf_getaiocap*() function provides the atomic I/O capabilities for the current mode (buffered or direct, etc.) of the file.

H.5.2.6 The Direct I/O Capabilities Buffer

There is no specific rationale for this subclause.

H.5.2.7 Which Argument Formats

A problem arises in the passage of the attributes in that some attributes (for example, initial allocation size) have no reasonable and simple way of specifying a default value, since both zero and the maximum value have meaning. For this reason, the structure of the attribute is always accompanied by a *which* parameter that specifies which of the elements of the structure are to be interpreted. Accordingly, attribute structures that are returned by the system are accompanied by a *which* parameter that specifies which of the attributes will be of interest to the application.

An example of returns in the which argument is as follows: an attempt to change the *atb_alloc* attribute may either change the actual allocation, in which case no indication is made in *actwhich* and *actattr*, or not change the allocation and return *atb_alloc* as the original reservation size, indicating in *actwhich* that *atb_alloc* was not changed.

H.5.2.8 Increment Lists

One type of data returned as a capability, namely allocation and extension increments, involves an additional complexity that is visible in the interface. In order to accommodate multiple models of storage, an increment type is provided, along with further interfaces, *rf_getallocincr*() and *rf_getincr*(). Three types of increments are acknowledged:

1) *Multiples*. The quantity is a number that must be supplied in an integral multiple, such as a file system that is optimized around 8 block allocation granules.
2) *Exponent*. The increment must be an exponential multiple of the increment size. For example, some memory disks may have optimal performance for 4, 16, 64 … block sizes and therefore would return an exponent 4 as the increment size.
3) *List*. The increment may also have nonlinear boundaries. This is actually the case for most disks, with disparate performance characteristics when crossing sector, track, and cylinder boundaries.

The *rf_getallocincr*() and *rf_getincr*() functions allows for item (3) by returning a list of possible increment sizes.

The *rf_getallocincr*() function is very similar to the *rf_getincr*() function but differs in the following respects. The *rf_getincr*() function returns an increment list for the various types of transfer sizes. These are of type *size_t*. The *rf_getallocincr*() function returns an increment list for the preallocation attributes of a file. These are of type *off_t*.

H.5.3 Realtime File Functions

An approach was initially considered that included three variations of each of the attribute/capability functions, one for an open file descriptor, one for a pathname, and one for the file system associated with a pathname. Due to difficulties associated with standardizing the definition of file system, the difficulty in some implementations of performing operations on a path rather than an open instance of the file, and the general complexity of this interface, the concept was dropped and the file-descriptor-only operations adopted.

Additionally, the original interface allowed the specification of an “intended attributes” buffer on capability functions. This would constrain the intent of the caller so that the capabilities returned by the system could be adjusted to show what was possible within that intent. Once again, this interface proved complex and cumbersome and was withdrawn.

H.5.3.1 Create a Realtime File

There is no specific rationale for this subclause.

H.5.3.2 Get Attributes of a Realtime File

There is no specific rationale for this subclause.

H.5.3.3 Set Attributes of a Realtime File

There is no specific rationale for this subclause.

H.5.3.4 Get Allocation Capabilities of Realtime File and File Systems

Again, complexity arguments justified the creation of four separate functions.

H.5.3.5 Get Cache Capabilities of Realtime Files and File Systems

There is no specific rationale for this subclause.

H.5.3.6 Get Buffered I/O Capabilities of Realtime Files and File Systems

There is no specific rationale for this subclause.

H.5.3.7 Get Atomic I/O Capabilities of Realtime Files and File Systems

There is no specific rationale for this subclause.

H.5.3.8 Get Direct I/O Capabilities of Realtime Files and File Systems

There is no specific rationale for this subclause.

H.5.3.9 Get Increment Lists

There is no specific rationale for this subclause.

H.5.3.10 Allocate Suitably Aligned Data Buffer

The *placebuf*() function was considered, but it raises several issues that need to be addressed.

The first of these is why *placebuf*() does not do the actual allocation of the buffer itself. The reason behind this is twofold. First, actual allocation is perceived to be a language binding issue, and in fact some languages such as FORTRAN have no allocation facility, while *malloc*() is defined in C Standard {2}. Thus other language bindings might be inconsistent. Second, it is often necessary to allocate a buffer in a place such as a particular shared memory region, which would be difficult to allow in a *malloc*()-derived allocation interface. Thus, the interface of choosing an aligned buffer from within the specified buffer was chosen.

After much debate, the interface was changed to include an actual allocator interface, *rf_getbuf*(), and a de-allocator, *rf_freebuf*(). This was deemed as acceptable due to the special requirements of this interface.

The second frequently encountered question is why an interface is needed when a compile time computation would suffice to do the calculation. Once again, the application may need to place a buffer in a shared memory region whose base alignment is not known to the application.

Note that the current cost of the allocation interface is the inability to place the buffer in shared memory.

Thirdly, it is not obvious how *placebuf*() can return an error. An example would be placing a 512 B buffer in a specified buffer that is 768 B but that itself begins at a multiple of 512 B + 4. In this case, no suitable alignment to satisfy the request exists.

H.5.3.11 Release a Previously Allocated Data Buffer

There is no specific rationale for this subclause.

H.5.4 Performance Metrics

The performance metrics given differ in structure from those given for many of the other interfaces in this part of ISO/IEC 9945 in that they actually measure performance of an I/O subsystem rather than the performance of the interface. The reason for this is that the requirement satisfied by the realtime files interface is satisfied in an indirect manner. To give an example, the goal of message passing is to send messages quickly, and therefore the important metric is the speed of the send message interface. In the case of realtime files, the speed of setting an attribute is basically unimportant. What is important is the speed of read and write operations to the file once the attributes are set. Hence, the supplied metrics give a set of examples that show how to obtain a bounded level of performance and what that level of performance may be expected to be.