INTERNATIONAL STANDARD

ISO/IEC 9945-1

IEEE Std 1003.1

First edition
1990-12-07

Information technology — Portable Operating System Interface (POSIX) —

Part 1 :
System Application Program Interface (API)
[C Language]

Technologies de l'information — Interface pour la portabilité des systèmes (POSIX) — Partie 1 : Interface programme de systèmes d'application (API) [Langage C]

Reference number
ISO/IEC 9945-1 : 1990 (E)
IEEE Std 1003.1-1990

ISBN 1-55937-061-0

Library of Congress Catalog Number 90-084554

Quote in 8.1.2.3 on Returns is taken from X3.159-1989, developed under the auspices of the American National Standards Accredited Committee X3 Technical Committee X3J11, Computer and Business Equipment Manufacturers Association (CBEMA), 311 First St, N.W., Suite 500, Washington, DC 20001.

International Standard ISO/IEC 9945-1: 1990
IEEE Std 1003.1-1990
(Revision of IEEE Std 1003.1-1988)

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

Sponsor
Technical Committee on Operating Systems and Application Environments of the IEEE Computer Society

Approved September 28, 1990
IEEE Standards Board

Approved 1990 by the
International Organization for Standardization
and by the
International Electrotechnical Commission

Abstract: ISO/IEC 9945-1: 1990 (IEEE Std 1003.1-1990), *Information technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language]* 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. This standard is stated in terms of its C binding.

Keywords: API, application portability, C (programming language), data processing, information interchange, open systems, operating system, portable application, POSIX, programming language, system configuration computer interface

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.**

IEEE Standards documents are developed within the Technical Committees of the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE that have expressed an interest in participating in the development of the standard.

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

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

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

Comments on standards and requests for interpretations should be addressed to:

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

Contents

PAGE

TABLES

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) together form a system for worldwide standardization as a whole. 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 approval before their acceptance as International Standards. They are approved in accordance with procedures requiring at least 75% approval by the national bodies voting.

International Standard ISO/IEC 9945-1: 1990 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*.

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* (under development)

— *Part 3: System administration* (under development)

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

International Organization for Standardization/International Electrotechnical Commission
Case postale 56 • CH-1211 Genève 20 • Switzerland

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], but is included for information only.)

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)

(4) Data interchange format (Section 10)

---

1) UNIX is a registered trademark of AT&T in the USA and other countries.

2) The vertical rules in the right margin depict technical or significant non-editorial changes from IEEE Std 1003.1-1988 to IEEE Std 1003.1-1990. A vertical rule beside an empty line indicates deleted text.

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 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

3) The IEEE is grateful to both AT&T and UniForum for permission to use their materials.

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 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.

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

Source, Not Object, Portability

This 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

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).

(3) Emulations hosted on entirely different operating systems

(4) Networked systems

(5) Distributed systems

(6) Systems running on a broad range of hardware

No direct references to this goal appear in this 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.

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) Realtime facilities

(6) Secure/Trusted System considerations

(7) Network interface facilities

(8) System Administration

(9) Graphical User Interfaces

(10) 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

(11) An overall guide to POSIX-based or related Open Systems standards and profiles

Extensions are approved as "amendments" or "revisions" to this document, following the IEEE and ISO/IEC Procedures.

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

If you have interest in participating in the TCOS 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 chairperson of the appropriate TCOS working group. If you have interest in participating in this work at the international level, contact your ISO/IEC national body.

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. Working drafts of POSIX standards under development are also available from this office.

IEEE Std 1003.1-1990 was prepared by the 1003.1 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 1003.1 Working Group was as follows:

Technical Committee on Operating Systems and Application Environments (TCOS)

Chair: Luis-Felipe Cabrera

Standards Subcommittee for TCOS

Chair: Jim Isaak
Treasurer: Quin Hahn
Secretary: Shane McCarron

1003.1 Working Group Officials

Chair: Donn Terry
Vice Chair: Keith Stuck
Editor: Hal Jespersen
Secretary: Keith Stuck

Working Group

Steve Bartels
Robert Bismuth
James Bohem
Kathy Bohrer
Keith Bostic
Jonathan Brown
Tim Carter
Myles Connors
Landon Curt Noll
Dave Decot
Mark Doran
Glenn Fowler
Greg Goddard
Andrew Griffith
Rand Hoven
Randall Howard
Mike Karels
Jeff Kimmel
David Korn
Bob Lenk
Shane McCarron
John Meyer
Martha Nalebuff
Neguine Navab
Paul Rabin
Seth Rosenthal
Lorne Schachter
Steve Schwarm
Paul Shaughnessy
Steve Sommars
Ravi Tavakley
Jeff Tofano
David Willcox
John Wu

The following persons were members of the 1003.1 Balloting Group that approved the standard for submission to the IEEE Standards Board:

David Chinn	*Open Software Foundation Institutional Representative*
Michael Lambert	*X/Open Institutional Representative*
Heinz Lycklama	*UniForum Institutional Representative*
Shane McCarron	*UNIX International Institutional Representative*

Helene Armitage
David Athersych
Timothy Baker
Geoff Baldwin
Steven E. Barber
Robert Barned
John Barr
James Bohem
Kathryn Bohrer
Robert Borochoff
Keith Bostic
James P. Bound
Joseph Boykin
Kevin Brady
Phyllis Eve Bregman
Fred Lee Brown, Jr.
A. Winsor Brown
Luis-Felipe Cabrera
Nicholas A. Camillone
Clyde Camp
John Carson
Steven Carter
Jerry Cashin
Kilnam Chon
Anthony Cincotta
Mark Colburn
Donald W. Cragun
Ana Maria DeAlvare
Dave Decot
Steven Deller
Terence Dowling
Stephen A. Dum
John D. Earls
Ron Elliott
David Emery
Philip H. Enslow
Ken Faubel
Kester Fong
Kenneth R. Gibb
Michel Gien
Gregory W. Goddard
Dave Grindeland
Judy Guist
James Hall
Charles Hammons
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

When the IEEE Standards Board approved this standard on September 28, 1990, it had the following membership:

Information technology—Portable operating system interface (POSIX)—Part 1: System application programming interface (API) [C Language]

Section 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

(8) The behavior of system services on systems supporting concurrency within a single process

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.

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.

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 this International Standard 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: 1983,[1] *Information processing—ISO 7-bit coded character set for information interchange.*

{2} ISO/IEC 9899: . . . ,[2] *Information technology—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) Under revision. (This notation is meant to explicitly reference the 1990 Draft International Standard version of ISO/IEC 646.)

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

2) To be approved and published.

(1) The system shall support all required interfaces defined within this part of ISO/IEC 9945. These 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 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}	Multiple groups option (in 2.8.3)
{_POSIX_JOB_CONTROL}	Job control option (in 2.9.3)
{_POSIX_CHOWN_RESTRICTED}	Administrative/security option (in 2.9.4)

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 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 routines 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, 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.

Section 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

(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

(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 a process.

2.2.2.4 appropriate privileges: An implementation-defined means of associating privileges with a 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 background process: A process that is a member of a background process group.

2.2.2.6 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.

2.2.2.7 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.

2.2.2.8 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.

2.2.2.9 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.

2.2.2.10 child process: See *process* in 2.2.2.62.

2.2.2.11 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*.

2.2.2.12 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.

2.2.2.13 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.

2.2.2.14 current working directory: See *working directory* in 2.2.2.89.

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

2.2.2.16 directory: A file that contains directory entries.

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

2.2.2.17 directory entry [link]: An object that associates a filename with a file.

Several directory entries can associate names with the same file.

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

See *pathname resolution* in 2.3.6.

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

See *pathname resolution* in 2.3.6.

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

2.2.2.21 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.

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

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

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

See *seconds since the Epoch*.

2.2.2.25 feature test macro: A `#define`d symbol used to determine whether a particular set of features will be included from a header.

See 2.7.1.

2.2.2.26 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 *FIFO*s are described in 5.3.1, 6.4.1, 6.4.2, and 6.5.3.

2.2.2.27 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.

2.2.2.28 file description: See *open file description* in 2.2.2.51.

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

2.2.2.30 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 group IDs of the process matches the group ID associated with the file. Other members of the class may be implementation defined.

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

2.2.2.32 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.

2.2.2.33 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.

2.2.2.34 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.

2.2.2.35 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.

2.2.2.36 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 2.3.2.

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

File serial numbers are unique throughout a file system.

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

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

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

2.2.2.40 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.

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

2.2.2.42 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.

2.2.2.43 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.

2.2.2.44 link: See *directory entry* in 2.2.2.17.

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

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

Each login shall be associated with exactly one login name.

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

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

See *file access permissions* in 2.3.2.

2.2.2.49 null string: See *empty string* in 2.2.2.23.

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

2.2.2.51 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.

2.2.2.52 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.

2.2.2.53 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.

2.2.2.54 parent process: See *process* in 2.2.2.62.

2.2.2.55 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.

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

2.2.2.57 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.

2.2.2.58 pathname component: See *filename* in 2.2.2.32.

2.2.2.59 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.

2.2.2.60 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.

2.2.2.61 privilege: See *appropriate privileges* in 2.2.2.4.

2.2.2.62 process: An address space and single thread of control that executes within that address space, and its required system resources.

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.

2.2.2.63 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.

2.2.2.64 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.

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

2.2.2.66 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.

2.2.2.67 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.

2.2.2.68 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. Its thread of control and 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.

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

2.2.2.70 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 2.2.2.42. This value is subject to change during the process lifetime, as described in 4.2.2.

2.2.2.71 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 2.2.2.87. This value is subject to change during the process lifetime, as described in 4.2.2.

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

2.2.2.73 relative pathname: See *pathname resolution* in 2.3.6.

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

2.2.2.75 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.

2.2.2.76 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.

2.2.2.77 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:

$$tm\_sec + tm\_min*60 + tm\_hour*3\,600 + tm\_yday*86\,400 +$$
$$(tm\_year-70)*31\,536\,000 + ((tm\_year-69)/4)*86\,400$$

2.2.2.78 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.

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

2.2.2.80 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.

2.2.2.81 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. The term *signal* is also used to refer to the event itself.

2.2.2.82 slash: The literal character "/".

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

2.2.2.83 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.

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

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

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

2.2.2.87 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.

2.2.2.88 user name: A string that is used to identify a user, as described in 9.1.

2.2.2.89 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, *Information technology—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

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.

2.4 Error Numbers

Most functions provide an error number in the external variable *errno*, which is defined as:

```
extern int errno;
```

The value of this variable shall be defined only after a call to a function for which it is explicitly stated to be set and until it is changed by the next function call. The variable *errno* should only be examined when it is indicated to be valid by a function's return value. No function defined in this part of ISO/IEC 9945 sets *errno* to zero to indicate an error.

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.

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).

[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.

[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-defined maximum file size.

[EINTR] Interrupted function call
An asynchronous signal (such as SIGINT or SIGQUIT; see the description of header `<signal.h>` in 3.3.1) 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}.

[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.

[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.

[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.

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.
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.

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_SOURCE.

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*"; *name*s 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 user's initial working directory from the user database (see the description of the header `<pwd.h>` in 9.2.2).

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 *name*s 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 *value*s 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 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.

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>`.

Table 2-2 – Reserved Header Symbols

Header	Key	Reserved Prefix	Reserved Suffix
<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]	
<pwd.h>	1	pw_	
<signal.h>	1	sa_	
	2	SIG_	
	2	SA_	
<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]	
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 `#define`d 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.

The following feature test macro is defined:

Name	Description
_POSIX_SOURCE	When an application includes a header described by POSIX.1, and when this feature test macro is defined according to the preceding rules: (1) All symbols required by POSIX.1 to appear when the header is included shall be made visible. (2) Symbols that are explicitly permitted, but not required, by POSIX.1 to appear in that header (including those in reserved namespaces) may be made visible. (3) Additional symbols not required or explicitly permitted by POSIX.1 to be in that header shall not be made visible.

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 reservation of 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 _POSIX_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_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_SOURCE) will be visible. Symbols from the namespace reserved for the implementation, as defined by the C Standard {2}, are also permitted. The symbols beginning with two underscores are examples of this.

If _POSIX_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 functions not returning a "plain" *int*.

For functions described in the C Standard {2} and included by reference in Section 8 (whether or not they are further described in this part of ISO/IEC 9945), these prototypes or declarations (if required) shall appear in the headers defined for them in the C Standard {2}. For other functions in this part of ISO/IEC 9945, the prototypes or declarations shall appear in the headers listed below. If a function is defined by this part of ISO/IEC 9945, is not described 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 function declared in it, whether or not it is mentioned in the Synopsis subclause for that function. The requirements about the visibility of symbols in 2.7.2 shall be honored.

`<dirent.h>`	*opendir*( ), *readdir*( ), *rewinddir*( ), *closedir*( )
`<fcntl.h>`	*open*( ), *creat*( ), *fcntl*( )
`<grp.h>`	*getgrgid*( ), *getgrnam*( )
`<pwd.h>`	*getpwuid*( ), *getpwnam*( )
`<setjmp.h>`	*sigsetjmp*( ), *siglongjmp*( )
`<signal.h>`	*kill*( ), *sigemptyset*( ), *sigfillset*( ), *sigaddset*( ), *sigdelset*( ), *sigismember*( ), *sigaction*( ), *sigprocmask*( ), *sigpending*( ), *sigsuspend*( )
`<stdio.h>`	*ctermid*( ), *fileno*( ), *fdopen*( )
`<sys/stat.h>`	*umask*( ), *mkdir*( ), *mkfifo*( ), *stat*( ), *fstat*( ), *chmod*( )
`<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>`	*time*( ), *tzset*( )
`<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 not 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.

2.8.3 Run-Time Increasable Values

The magnitude limitations in Table 2-4 shall be fixed by specific implementations.

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.[3] 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.

3) 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 2-3 – Minimum Values

Name	Description	Value
{_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_LINK_MAX}	The value of a file's link count.	8
{_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_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_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_TZNAME_MAX}	The maximum number of bytes supported for the name of a time zone (not of the **TZ** variable).	3

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}

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.

Table 2-5 – Run-Time Invariant Values (Possibly Indeterminate)

Name	Description	Minimum Value
{ARG_MAX}	Maximum length of arguments for the *exec* functions, in bytes, including environment data.	{_POSIX_ARG_MAX}
{CHILD_MAX}	Maximum number of simultaneous processes per real user ID.	{_POSIX_CHILD_MAX}
{OPEN_MAX}	Maximum number of files that one process can have open at any given time.	{_POSIX_OPEN_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}
{TZNAME_MAX}	The maximum number of bytes supported for the name of a time zone (not of the **TZ** variable).	{_POSIX_TZNAME_MAX}

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.

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}

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.

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}

2.9 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 2-8. 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.

2.9.2 Symbolic Constant for the *lseek*() Function

The constants used by the *lseek*() function are shown in Table 2-9.

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.

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.

Table 2-10 – Compile-Time Symbolic Constants

Name	Description
{_POSIX_JOB_CONTROL}	If this symbol is defined, it indicates that the implementation supports job control.
{_POSIX_SAVED_IDS}	If defined, each process has a saved set-user-ID and a saved set-group-ID.
{_POSIX_VERSION}	The integer value 199009L. This value shall be used for systems that conform to this part of ISO/IEC 9945.

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.

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_CHOWN_RESTRICTED}	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_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.

Section 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.)

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 *const 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.

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).

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.

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; *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.

3.2 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).

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 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 process until status information for one of its terminated child processes is available, or until a signal whose action is either to execute a signal-catching function or to terminate the process is delivered. 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. Otherwise, its behavior shall be modified by the values of the *pid* and *options* arguments.

The *pid* argument specifies a set of child processes for which status is requested. The *waitpid*() function shall only return the status of a child process from this set.

(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 process 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).

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 *waitpid*() 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.

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.

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.

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 shall be defined by all implementations. However, implementations that do not support job control are not required to support these signals. 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.

3.3.1.2 Signal Generation and Delivery

A signal is said to be *generated* for (or sent to) a process when the event that causes the signal first occurs. Examples of such events include detection of hardware faults, timer expiration, and terminal activity, as well as the invocation of the *kill*() function. In some circumstances, the same event generates signals for multiple processes.

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.

During the time between the generation of a signal and its delivery, the signal is said to be *pending*. Ordinarily, this interval cannot be detected by an application. However, a signal can be *blocked* from delivery to a process. If the action associated with a blocked signal is anything other than to ignore the signal, and if that

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.

signal is generated for the process, the signal shall remain pending until either it is unblocked or the action associated with it is set to ignore the the signal. If the action associated with a blocked signal is to ignore the signal, and if that signal is generated for the process, it is unspecified whether the signal is discarded immediately upon generation or remains pending.

Each process has a *signal mask* that defines the set of signals currently blocked from delivery to it. The signal mask for a process is initialized from that of its parent. The *sigaction*(), *sigprocmask*(), and *sigsuspend*() functions control the manipulation of the signal mask.

The determination of which action is taken in response to a signal is made at the time the signal is delivered, allowing for any changes since the time of generation. This determination is independent of the means by which the signal was originally generated. If a subsequent occurrence of a pending signal is generated, it is implementation defined as to whether the signal is delivered more than once. The order in which multiple, simultaneously pending signals are delivered to a process is unspecified.

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

An implementation shall document any 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 Table 3-1 and Table 3-2.

(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.

(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, or SIGSEGV signal that was not generated by the *kill*() function or the *raise*() function 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.

(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) The signal-catching function shall be entered as a C language function call as follows:

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

where *func* is the specified signal-catching function and *signo* is the signal number of the signal being delivered.

(c) The behavior of a process is undefined after it returns normally from a signal-catching function for a SIGFPE, SIGILL, or SIGSEGV signal that was not generated by the *kill*() function or the *raise*() function 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 reentrant with respect to signals (that is, applications may invoke them, without restriction, from signal-catching functions).

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

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.

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, either *sig* or at least one pending unblocked signal shall be delivered to the sending process 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.

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).

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 *sigaddset*(), *sigdelset*(), *sigismember*(), *sigaction*(), *sigprocmask*(), *sigpending*(), or *sigsuspend*() 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; *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.

Implementations may add extensions as permitted in 1.3.1.1, point (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. The *sa_handler* field of the *sigaction* structure identifies the action to be associated with the specified signal. If the *sa_handler* field specifies a signal-catching function, the *sa_mask* field identifies a set of signals that shall be added to the signal mask of the process 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 bit, defined in the header `<signal.h>`, can be set in *sa_flags*:

Symbolic Constant	Description
SA_NOCLDSTOP	Do not generate SIGCHLD when children stop.

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.

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 *sigsuspend*() 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].

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.

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; *sigprocmask*(), 3.3.5; *sigsetops*, 3.3.3; *sigsuspend*(), 3.3.7.

3.3.5 Examine and Change Blocked Signals

Function: *sigprocmask*()

3.3.5.1 Synopsis

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

3.3.5.2 Description

The *sigprocmask*() function is used to examine or change (or both) the signal mask of the calling 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 process 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, or SIGSEGV signals are generated while they are blocked, the result is undefined, unless the signal was generated by a call to the *kill*() function or the *raise*() function defined by the C Standard {2}.

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

3.3.5.3 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.

3.3.5.4 Errors

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

[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; *sigpending*(), 3.3.6; *sigsetops*, 3.3.3; *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 the set of signals that are blocked from delivery and pending for the calling process in the space pointed to by the argument *set*.

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; *sigprocmask*(), 3.3.5; *sigsetops*, 3.3.3.

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 process with the set of signals pointed to by the argument *sigmask* and then suspends the process 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 *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; 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; *sigpending*(), 3.3.6; *sigprocmask*(), 3.3.5; *sigsetops*, 3.3.3.

3.4 Timer Operations

A process can suspend itself for a specific period of time with the *sleep*() function or suspend itself indefinitely with the *pause*() function until a signal arrives. The *alarm*() function schedules a signal to arrive at a specific time, so a *pause*() suspension need not be indefinite.

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 send the calling process a SIGALRM signal after the number of real-time 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.

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

3.4.2.4 Errors

If any of the following conditions occur, the *pause*() 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.

3.4.2.5 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 process to be suspended from execution until either the number of real-time seconds specified by the argument *seconds* have elapsed or a signal is delivered to the calling process 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.

Section 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 supplemental 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 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 and set *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*()

4.2.4.1 Synopsis

```
char *getlogin(void);
```

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.

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.

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.

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 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 process ID of this terminated child. See 3.2.1. If a child process has not waited 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_utime*s and *tms_cutime*s of the child processes.

The value *tms_cstime* is the sum of the *tms_stime*s and *tms_cstime*s 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.

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*(), *isatty*()

4.7.2.1 Synopsis

```
char *ttyname(int fildes);
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.

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.

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.

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
{ARG_MAX}	{_SC_ARG_MAX}
{CHILD_MAX}	{_SC_CHILD_MAX}
clock ticks/second	{_SC_CLK_TCK}
{NGROUPS_MAX}	{_SC_NGROUPS_MAX}
{OPEN_MAX}	{_SC_OPEN_MAX}
{STREAM_MAX}	{_SC_STREAM_MAX}
{TZNAME_MAX}	{_SC_TZNAME_MAX}
{_POSIX_JOB_CONTROL}	{_SC_JOB_CONTROL}
{_POSIX_SAVED_IDS}	{_SC_SAVED_IDS}
{_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.

Section 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*(), *rewinddir*(), *closedir*()

5.1.2.1 Synopsis

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *dirname);

struct dirent *readdir(DIR *dirp);

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.

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*() 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.

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 a value of **NULL** 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, when the condition is detected, the *opendir*() function shall return a value of **NULL** 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, when the condition is detected, the *readdir*() function shall return a value of **NULL** 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, when 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.

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 pathname.

[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_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 until a process opens the file for writing. An *open*() for writing-only shall block until a process 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 wait until the device is ready or available before returning.

(3) Otherwise, the behavior of O_NONBLOCK is unspecified.

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 O_TRUNC is set and the file did previously exist, upon successful completion the *open*() function shall mark for update the *st_ctime* 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.

[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; *stat*(), 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/stat.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 Name	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.

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_ctime* Changed file status, for example, *chmod*().

These times are updated as described in 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*.

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 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 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 and set *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*()

5.6.4.1 Synopsis

```
#include <sys/types.h>
#include <sys/stat.h>

int chmod(const char *path, 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 2.3.2. Additional implementation-defined restrictions may cause the S_ISUID and S_ISGID bits in *mode* to be ignored.

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, 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*().

The effect on file descriptors for files open at the time of the *chmod*() function is implementation defined.

Upon successful completion, the *chmod*() 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.

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 (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] The effective user ID does not match the owner of the file, or the calling process does not have appropriate privileges and {_POSIX_CHOWN_RESTRICTED} indicates that such privilege is required.

[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, point (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 and set *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.

5.7 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*.

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*.

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_CHOWN_RESTRICTED}	{_PC_CHOWN_RESTRICTED}	(7)
{_POSIX_NO_TRUNC}	{_PC_NO_TRUNC}	(3, 4)
{_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.

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.

Section 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.

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.

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 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 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.

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 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 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) shall be 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 process 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 with *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 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.

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, ...);
```

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.

Table 6-5 – File Status Flags Used for *open*() and *fcntl*()

Constant	Description
O_APPEND	Set append mode.
O_NONBLOCK	No delay.

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.

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 the 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.

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 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 process 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.

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

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.

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.

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.

Otherwise, a value of –1 shall be returned and *errno* shall be set to indicate the error.

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.

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.

[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.

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.

Section 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 of 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 in 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	Subscript Usage: 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	19 200 baud
B300	300 baud	B38400	38 400 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.[4)]

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.

4) 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.

(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.

Section 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: setjmp, longjmp.

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 2.2.2.77, 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*().

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 ($0 \leq d \leq 6$) 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.

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.

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_ctime* 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*().

8.3 Other C Language Functions

8.3.1 Nonlocal Jumps

Functions: *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 process (see 3.3.1) 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.

8.3.1.3 Cross-References

sigaction(), 3.3.4; <signal.h>, 3.3.1; *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.

Section 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*(), *getgrnam*()

9.2.1.1 Synopsis

```
#include <sys/types.h>
#include <grp.h>

struct group *getgrgid(gid_t gid);

struct group *getgrnam(const char *name);
```

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.

9.2.1.3 Returns

A NULL pointer is returned on error or if the requested entry is not found.

The return values may point to static data that is overwritten by each call.

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.

9.2.1.5 Cross-References

getlogin(), 4.2.4.

9.2.2 User Database Access

Functions: *getpwuid*(), *getpwnam*()

9.2.2.1 Synopsis

```
#include <sys/types.h>
#include <pwd.h>

struct passwd *getpwuid(uid_t uid);

struct passwd *getpwnam(const char *name);
```

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.

9.2.2.3 Returns

A NULL pointer is returned on error or if the requested entry is not found.

The return values may point to static data that is overwritten on each call.

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.

9.2.2.5 Cross-References

getlogin(), 4.2.4.

Section 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 Annex 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     04000   /* Set UID on execution */
#define TSGID     02000   /* Set GID on execution */
#define TSVTX     01000   /* Reserved */
                          /* File permissions */
```

```
#define TUREAD   00400   /* Read by owner */
#define TUWRITE  00200   /* Write by owner */
#define TUEXEC   00100   /* Execute/Search by owner */
#define TGREAD   00040   /* Read by group */
#define TGWRITE  00020   /* Write by group */
#define TGEXEC   00010   /* Execute/Search by group */
#define TOREAD   00004   /* Read by other */
#define TOWRITE  00002   /* Write by other */
#define TOEXEC   00001   /* 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 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 646 {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 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 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 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; `<pwd.h>`, 9.2.2; `<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

Field Name	Header 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
Field Name	**File Name Length**	**Interpreted as**
c_name	*c_namesize*	Pathname string
Field Name	**File Data 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; `<pwd.h>`, 9.2.2; `<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.

Annex A
(informative)
Bibliography

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.*[1)]

{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.*[2)]

{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.*

---

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

2) IEC documents can be obtained from the IEC office, 3, rue de Varembé, Case Postale 131, CH-1211, Genève 20, Switzerland/Suisse.

{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.*

{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.*[3]

{B20} CCITT Recommendation X.212, *Information processing systems—Data communication—Data link service definition for Open Systems Interconnection.*

3) CCITT documents can be obtained from the CCITT General Secretariat, International Telecommunications Union, Sales Section, Place des Nations, CH-1211, Genève 20, Switzerland/Suisse.

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[4], *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.*

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: . . . ,[5] *Information technology—Portable operating system interface (POSIX)—Part 2: Shell and utilities.*

---

4) ANSI documents can be obtained from the Sales Department, American National Standards Institute, 1430 Broadway, New York, NY 10018.

5) To be approved and published.

{B37} ISO/IEC 10646: . . . ,[6] *Information processing—Multiple octet coded character set.*

{B38} IEEE Std 100-1988, *IEEE Standard Dictionary of Electrical and Electronics Terms.*

A.3 Historical Documentation and Introductory Texts

{B39} American Telephone and Telegraph Company. *System V Interface Definition (SVID), Issues 2 and 3.* Morristown, NJ: UNIX Press, 1986, 1989.[7]

{B40} American Telephone and Telegraph Company. *UNIX System III Programmer's Manual.* Greensboro, NC: Western Electric Company, October 1981.

{B41} American Telephone and Telegraph Company. *UNIX Time Sharing System: UNIX Programmer's Manual.* 7th ed. Murray Hill, NJ: Bell Telephone Laboratories, January 1979.

{B42} "The UNIX System."[8] *AT&T Bell Laboratories Technical Journal.* vol. 63 (8 Part 2), October 1984.

{B43} "UNIX Time-Sharing System."[9] *Bell System Technical Journal.* vol. 57 (6 Part 2), July-August 1978.

{B44} Bach, Maurice J. *The Design of the UNIX Operating System.* Englewood Cliffs, NJ: Prentice-Hall, 1987.

{B45} Harbison, Samuel P. and Steele, Guy L. *C: A Reference Manual.* Englewood Cliffs, NJ: Prentice-Hall, 1987.

{B46} Kernighan, Brian W. and Ritchie, Dennis M. *The C Programming Language.* Englewood Cliffs, NJ: Prentice-Hall, 1978.

{B47} Kernighan, Brian W. and Pike, Rob. *The UNIX Programming Environment.* Englewood Cliffs, NJ: Prentice-Hall, 1984.

{B48} 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.

{B49} McGilton, Henry and Morgan, Rachel. *Introducing the UNIX System.* New York: McGraw-Hill (BYTE Books), 1983.

6) To be approved and published.

7) 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.

8) This entire edition is devoted to the UNIX system.

9) This entire edition is devoted to the UNIX time-sharing system.

{B50} Organick, Elliot I. *The Multics System: An Examination of Its Structure.* Cambridge, MA: The MIT Press, 1972.

{B51} 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.

{B52} Ritchie, Dennis M. "Reflections on Software Research." *Communications of the ACM.* vol. 27 (8), August 1984, pp. 758–760. ACM Turing Award Lecture.

{B53} Ritchie, Dennis. "The Evolution of the UNIX Time-Sharing System." *AT&T Bell Laboratories Technical Journal.* vol. 63 (8), October 1984, pp. 1577–1593.

{B54} 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.

{B55} 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.

{B56} Ritchie, Dennis M. "Unix: A Dialectic." *Winter 1987 USENIX Association Conference Proceedings, Washington, D.C.*, pp. 29–34. Berkeley, CA: USENIX Association, January 1987.

{B57} Rochkind, Marc J. *Advanced UNIX Programming.* Englewood Cliffs, NJ: Prentice-Hall, 1985.

{B58} 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.

{B59} /usr/group Standards Committee. *1984 /usr/group Standard.* Santa Clara, CA: UniForum, 1984.

{B60} X/Open Company, Ltd. *X/Open Portability Guide, Issue 2.* Amsterdam: Elsevier Science Publishers, 1987.

{B61} X/Open Company, Ltd. *X/Open Portability Guide, Issue 3.* Englewood Cliffs, NJ: Prentice-Hall, 1989.

Annex B
(informative)

Rationale and Notes

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.[1)]

B.1 Scope and Normative 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 POSIX.1's developers 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.

1) 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.

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 the 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 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, the Rationale includes some references to 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 standalone 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.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 {B46} or the *1984 /usr/group Standard* {B59}. 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* {B59}), 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.1.2 Normative 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 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 optional job control feature, 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*.

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.2.3.7.

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 "536 457 599 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 job-control 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 {B41}.

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.

***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 job-control-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.

***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.2.3.5.

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.

***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.

***super-user:** This concept, with great historical significance to UNIX system users, has been replaced with the notion of *appropriate privileges*.

B.2.2.3 Abbreviations

There is no additional rationale provided for this subclause.

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.

B.2.3.5 filename portability: 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.2.3.6 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.2.3.7 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.2.4 Error Numbers

The definition of *errno* in POSIX.1 is stricter than that in the C Standard {2}. The C Standard {2} merely requires that it be an assignable *lvalue*. The POSIX.1 *extern int errno* meets that requirement and supports historical usage as well.

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.

[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.

[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 *siglongjmp*().

[ENOMEM] The term *main memory* is not used in POSIX.1 because it is implementation defined.

[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.2.5 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"[2)] 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.

2) 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."

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.

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 *int*s 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 Annex 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, _POSIX_SOURCE 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_SOURCE would be just another ordinary feature test macro. The only order requirement is "before headers.")

If _POSIX_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_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_SOURCE is not used, the implementation default, which is probably backwards compatible, determines their visibility.)

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_SOURCE is defined.)

— 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.

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.

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* {B39} 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.

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 *exec*s without explicit reason to do so, and especially not to block signals across *exec*s 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:

src: \ *dst:*	`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 {ARG_MAX} applies not just to the size of the argument list, but to the sum of that and the size of the environment list.

[EFAULT] Some historical systems return [EFAULT] rather than [ENOEXEC] when the new process image file is corrupted. They are 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.3.2 Process Termination

Early drafts 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 that 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* {B59}, 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.

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.

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*. The POSIX.1 definition of *delivery* does not exclude ignored signals; this is considered a more consistent definition.

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.

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 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.

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.

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 reasonable to initialize part of the structure, such as a version field, to permit binary compatibility between releases where the size of the set varies. For such reasons, either *sigemptyset*() or *sigfillset*() must be called prior to any other use of the signal set, even if such use is read-only [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*().

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.3.4 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 "real time" 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*.

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 "real time" 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 IEEE Std 1003.1-1990 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())
```

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* {B59} 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 are 60 clock ticks per second, or less than a year if there are 100. There are individual systems that run continuously for longer than that. 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.

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.

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* {B39} 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 after 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.

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* {B39}, 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 Process's Root Directory

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* {B59}, 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* {B59} was published, it contained [EEXIST] instead. When these functions were adopted into System V, the *1984 /usr/group Standard* {B59} 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 B.5.5 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* {B59} and is still in the *SVID* {B39}, 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 File 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 a 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 need read permission. An implementation that supports write-up operations could fail *fstat*() function calls even though it has a valid file descriptor open for writing.

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 the process's real user ID is 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.5.7 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* {B59} 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* {B59} 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* {B59})

(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 B.6.

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* {B59} 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* {B59}. 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.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* {B59} 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[3] 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

— It was concerned with setting and control of hardware timers.

— The appropriate timers and settings vary widely internationally.

— 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

3) 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."

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 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 intercharacter 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