IEEE Std 1003.1-1988

IEEE Standard Portable Operating System Interface for Computer Environments

POSIX

1003.1

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

SH12211

This standard has been adopted for Federal Government use.

Details concerning its use within the Federal Government are contained in Federal Information Processing Standards Publication 151-1, POSIX: Portable Operating System Interface for Computer Environments. For a complete list of publications available in the Federal Information Processing Standards Series, write to the Standards Processing Coordinator (ADP), National Computer Systems Laboratory, National Institute of Standards and Technology, Gaithersburg, MD 20899.

IEEE Standard Portable Operating System Interface for Computer Environments

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 which 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 once 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 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 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
345 East 47th Street
New York, NY 10017
USA

IEEE Std 1003.1-1988
(Revision of IEEE Std 1003.1
issued for Trial-Use in April 1986)

IEEE Standard Portable Operating System Interface for Computer Environments

Sponsor

**Technical Committee on Operating Systems
of the
IEEE Computer Society**

Approved August 22, 1988

IEEE Standards Board

Approved November 10, 1989

American National Standards Institute

Abstract: ANSI/IEEE Std 1003.1, *IEEE Standard Portable Operating System Interface for Computer Environments*, 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: applications interface to basic system services, applications for open systems, data interchange format, open systems, portable operating system interface for computer environments, user interfaces to open systems

Third Printing
June 1990

ISBN 1-55937-003-3

Library of Congress Catalog Number 88-082605

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

September 30, 1988 *SH12211*

Foreword

(This Foreword is not a part of IEEE Std 1003.1-1988, IEEE Standard Portable Operating System Interface for Computer Environments.)

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

In its present form, the standard focuses primarily on the C Language interface to the operating system.

IEEE Std 1003.1-1988 is the first of a group of proposed standards known colloquially, and collectively, as POSIX†. The other POSIX standards are described in Appendix A.

Organization of the Standard. The standard is divided into four parts:

(1) Statement of scope (Chapter 1)
(2) Definitions and global concepts (Chapter 2)
(3) The various interface facilities (Chapters 3 through 9)
(4) Data interchange format (Chapter 10)

This foreword and the appendices are not considered part of the standard.

Most of the sections describe a single service interface. The C Language binding for the service interface is given in the subsection labeled **Synopsis**. The **Description** subsection 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** subsections specifying return values and possible error conditions. **References** are used to direct the reader to other related sections. Additional material to complement sections in the standard may be found in **Rationale and Notes**, Appendix B. This appendix provides historical perspectives into the technical choices made by the 1003.1 Working Group. It also provides information to emphasize consequences of the interfaces described in the corresponding section of the standard.

In publishing this standard, both the IEEE and the 1003.1 Working Group simply intend to provide a yardstick against which various operating system implementations can be measured for conformance. It is *not* the intent of either the IEEE or the 1003.1 Working Group to measure or rate any products, to reward or sanction any vendors of products for conformance or lack of conformance to this standard, or to attempt to enforce this standard by these or any other means. The responsibility for determining the degree of conformance or lack thereof with this standard rests solely with the individual who is evaluating the product claiming to be in conformance with the standard. (See **Verification Testing** §A.2.4 for additional information on this subject.)

---

* UNIX is a registered trademark of AT&T.

† POSIX is pronounced *pahz-icks*, similar to *positive*.

Base Documents. The various interface facilities described herein are based on the *1984 /usr/group Standard* derived and published by the /usr/group Standards Committee, Santa Clara, California. The *1984 /usr/group Standard* and subsequent work of the 1003.1 Working Group is largely based on UNIX Seventh Edition, UNIX System III, UNIX System V, 4.2BSD, and 4.3BSD documentation, but wherever possible, compatibility with other systems derived from the UNIX operating system, or systems compatible with that system, has been maintained.

The IEEE is grateful to both AT&T and /usr/group for permission to use their materials.

Typographic Conventions. This standard uses the following typographic conventions:

(1) The *italic* font is used for the initial appearances of defined terms; cross references to defined terms within the sections **Terminology** §2.1, **Conformance** §2.2, and **General Terms** §2.3; symbolic parameters that are generally substituted with real values by the application; C language data types and function names (except in function **Synopsis** sections); and global external variable names.

(2) The **bold** font is used for C language data types and function names within function **Synopsis** sections; references to other sections or chapters within the standard. A bold word in all capital letters, such as

PATH

represents an environment variable, as described in **Environment Description** §2.7.

(3) The `constant-width` font is used to illustrate examples of system input or output where exact usage is depicted. It is also used for references to utility names defined in IEEE Std 1003.2 or found in historical implementations.

(4) Error number values returned by many functions are represented as:

[ERRNO]

See **Error Numbers** §2.5.

(5) Symbolic constants or limits defined in certain headers are represented as:

{LIMIT}

See **Numerical Limits** §2.9 and **Symbolic Constants** §2.10.

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

Extensions and Supplements to this Standard. Activities to extend this standard to address additional requirements are in progress and similar efforts can be anticipated in the future. This is an outline of how these extensions will be incorporated, and also how users of this document can keep track of that status.

Extensions are approved as *Supplements* to this document, following the IEEE Standards Procedures.

Approved Supplements are published separately and distributed with orders from the IEEE for this document until the full document is reprinted and such supplements are incorporated in their proper positions.

If you have any question about the completeness of your version, you may contact the IEEE Computer Society [(202) 371-0101] or the IEEE Standards Office [(212) 705-7960] to determine what supplements have been published.

Supplements are numbered in the same format as the main document, and with unique positions as either subsections or main sections. A supplement may include new subsections in various sections of the main document as well as new main sections. Supplements may include new sections in already approved supplements. However, the overall numbering shall be unique so that two supplements do not use the same numbers unless one replaces the other.

Supplements may contain either required functions or optional facilities. Supplements may add additional conformance requirements (see **Conformance** §2.2) defining new classes of conforming systems or applications.

It is undesirable, but perhaps unavoidable, for supplements to change the functionality of the already defined facilities.

Supplements are not used to provide a general update of the standard. This is done through the review procedure as specified by the IEEE.

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

(1) Shell and Utility facilities — P1003.2 (see **Shell and Utilities** §A.2.3);

(2) Verification Testing — P1003.3 (see **Verification Testing** §A.2.4);

(3) Realtime facilities — P1003.4 (see **Realtime Extensions** §A.2.5);

(4) Ada* Language bindings — P1003.5 (see **Ada Language Bindings** §A.2.6);

(5) Secure/Trusted System considerations — P1003.6 (see **Trusted System Extensions** §A.2.7);

(6) Language-independent service descriptions (this will be the subject of a future supplement to this standard);

(7) FORTRAN Language bindings;

(8) Network interface facilities (see **Networking Standards** §A.2.10);

(9) System Administration (see **System Administration Extensions** §A.2.9);

(10) An overall guide to POSIX-based or related Open Systems standards — P1003.0 (see **Open System Guidelines** §A.2.8).

If you have interest in participating in the working groups addressing these issues, please send your name, address, and phone number to the:

---

* Ada is a registered trademark of the U.S. Government — Ada Joint Program Office.

Secretary, IEEE Standards Board
Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street
New York, NY 10017
USA

and ask to have this forwarded to the chairperson of the appropriate 1003 Working Group.

IEEE Std 1003.1-1988 was prepared by the 1003.1 Working Group, sponsored by the Technical Committee on Operating Systems 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 (TCOS)

Chair: Joseph Boykin

Standards Subcommittee for TCOS

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

1003.1 Working Group Officials

Chair:	Jim Isaak
Co-Chair:	Donn Terry
	Heinz Lycklama (1985-1986)
Editor:	Hal Jespersen
	Jim McGinness (1985-1986)
Secretary:	Shane McCarron
	Stephen Head (1985-1986)

Working Group

Linda Abelu
Robert Adams
Noboru Akima
Bill Allen
Pat Amaranth
Dennis K. Anderson
Wolf Arfvidson
Arun K. Arya
Karl Auerbach
Jeanne Baccash
Brian Baird
Jayne Baker
Karen Baker
Geoff Baldwin
Mike Banahan
Jerome Banasik
John Barr
A. L. Barrese
Stephen R. Bartels
John-Olof Bauner
M. T. Carrasco Benitez
David Bernstein
Cynthia Berthold
Craig Bevins
David Bird
Andy Bishop
John A. Black
Ed Blackmond
Betsy Blankenhorn
Jim Blondeau
Kathy Bohrer
Cornelia Boldyreff
William D. Bone
Michael Bordelon
William Borkowski
David Borman
Robert Borochoff
Paul L. Borrill
Keith Bostic
James P. Bound
Claude Bourstin
Stowe Boyd
Joseph Boykin
Brian Boyle
Kevin Brady
Joe B. Brame
Phyllis Eve Bregman
Sven Brehmer
Stephen J. Brodeur
James J. Brodie
Peter Brouwer
Robert A. Brown
Warren Brown
J. David Bullis
Steve Bunch
Clarice M. Burch
Steven J. Buroff
Raymond Burret
Bruce Calkins
George Cameron

N. A. (Nick) Camillone
John R. Campbell
Paul Cantrell
James A. Capps
John Carmichael
Lisa Carnahan
Bill Carpenter
Stephen E. Carpenter
John H. Carson
Donald J. Carter
Steven. L. Carter
T. J. Cashin
John Caywood
Chin Chao
K. Herman Chen
Kilnam Chon
Chan Fung Chong
Lino Chung
Anthony V. Cincotta
Robert Claeson
Richard Clark
Paul Clarke
David A. Cobb
Mark Colburn
Clement T. Cole
N. C. Comsudi
Peter Cook
Bill Corwin
Mike Cossey
William T. Cox
Roy A. Crabtree
Gloria Cracovia
Donald W. Cragun
Allen Crawley
Steve Crise
George C. Dalmas III
Ajit Dandadani
James A. Davis
James R. Davis
Steven Davis
Dave Decot
William DeKeyser
Steven R. Deller
Craig W. DeNoce
Robert J. Devine
Heinz Diehl
Steve Diller
G. C. Dimitriou
David L. Dodge
John Douglas
Terence S. Dowling
Tony Downes
Pat Dukes
Stephen Dum
Dominic Dunlop
Larry Dwyer
William J. Eagan
Michael A. Edmonds
Ron Elliott
Dave Emery
D. H. G. Epema
Fran Fadden
Allen Farris
Maurice Fathi
Kenneth T. Faubel
A. Michael Ferris
Don E. Folland
Peter Fonash
Kester Fong
Martin C. Fong
Terence Fong
Belinda Frazier
Art Fritzson
Alan Fryer
Mitchell Fuchs
Mark Funkenhauser
Thomas S. Gary
Louise Germani
Michael Gersten
John Gertwagen
Al Gettier
Usman A. Ghani
Kenneth R. Gibb
Michel Gien
Alton L. Gilbert
Lloyd E. Gilbert
Timothy D. Gill
Steve Glaser
Stuart Glickman
Jeff Goldberg
Mitch Gort
Loretta Goudie
Randall W. Graves
John Gregg
William F. Griffeth Jr.
Robert C. Groman
Judy Guist
Mesut Gunduc
Douglas A. Gwyn
Quin Hahn
Bob Hairfield
Richard F. Hamm
Richard Hammons
Allen L. Hankinson
John Hanley
Tony Hansen
Dale Harris
Mike Haskell
Paul Haskell
Elaine Hauser
Terry Hayes
Stephen M. Head
Barry E. Hedquist
Henry G. Hefferman
Terry Heidelberg
Hans H. Heilborn
Amy Helmers
Johan Helsingius
Michael Herman
John Hesterberg
Wolfgang B. Hofs
Leon M. Holmes
D. Ian Hopper
Thomas F. Houghton
Jim Houston
Rand A. Hoven
John Howard
Randall Howard
Michael J. Hsu
Cheng Hu
Andrew R. Huber
Jan Huffman
Chris Hughes
Glenn C. Hughes
Gerald L. Ingalls
James D. Isaak
Keld Jorn iSimonsen
Doug Jacobson
Steven A. James
Steve Jennings
Hal Jespersen
Greg Jones
Christopher Juillet
Derek C. Kaufman
Sol Kavy
Gregg Kellogg
Gregory Kenley
Francis X. Kenney
Jerry Keselman
Lorraine C. Kevra
Karl Kimball
Jeffrey S. Kimmel
Dale L. Kirkland
Yvon Klein
Robert Kleinschmidt
Brad Kline
David Kline
John T. Kline
Kenneth C. Klingman
Joshua W. Knight
David Korn
John Krause
Donald Kretsch
Peter Krupp
Geoff Kuenning
D. Richard Kuhn
Takahiko Kuki
Anne M. Kumor
Dan Ladermann
John R. LaLonde
Lak Ming Lam

J. Eli Lamb
Mike G. Lambert
A. T. Landberg
Larry D. Landis
William Laplant
Steve Law
Benjamin Laws
Clifford D. Layton
Sue Le Grand
Doris Lebovits
Jeff Lee
Maggie Lee
Perry Lee
Sue Legrand
Greger K. Leijonhufvud
Peter Lemkin
Robert Lenk
David Lennert
Bob Lent
Thomas J. Leonard
Kin L. Leung
Kevin Lewis
Ben Liao
Y. K. Liu
Ben Livson
C. Douglass Locke
John Lomas
Julian Lomberg
James P. Lonjers
Warren E. Loper
Brian G. Lucas
Robert D. Luken
Craig Lund
Heinz Lycklama
Rod MacDonald
Robert J. Makuwski
Michael A. Marciniszyn
Benson I. Margulies
Anthony J. Mark
Bruce A. Martin
Roger Martin
Shane P. McCarron
Brian S. McCarthy
James McGinness
Marty McGowan
Marshall Kirk McKusick
M. R. Meaden
James C. Mechtel
Sunil Mehta
Paul Merry
Paul Metz
Bill Meyers
Bill Middlecamp
Gary Miller
Hugh Th. Miller
Conrad Minshall
Eugene Miya
Jim Moe
William Moloney
James Mooney
Cliff Moore
Jim Moore
Kathy Morgan
Rockie Morgan
Chalmers J. Morris
Gerry Morrone
Brian M. Morton
James Moseman
Paul Moskowitz
James Muehlen
Diane Mularz
Lance Murray
Joseph C. Musacchia
Narendran Nachiappan
Martha Nalebuff
Hirokazu Narita
Matt Narotam
Bret Needle
Sonya D. Neufer
Landon Curt Noll
Peter Norwood
Fred Noz
Alan F. Nugent
Greg Nuss
Karl Nyberg
Michael D. O'Dell
Robin T. O'Neill
Timothy Lynn Oglesby
Hagai Ohel
Gary M. Oing
Jim R. Oldroyd
Daniel Owens
Mark Parenti
Thomas J. Parenty
Bob Parlock
Gordon R. Parry
Marilyn F. Partel
Bhupendra Patel
Curt A. Paulson
Robert R. Peglar
Jon J. Penner
John C. Penney
Frank Perron
Patrick W. Peters
Donald A. Peterson
Jeffrey Picciotto
Gilbert W. Pilz
Gerald Powell
John J. Puttress
John S. Quarterman
Wendy Rauch-Hindin
Carol L. Raye
Rob Redford
Lizabeth Reilly
Michael P. Ressler
Phil Reston
Roberto Ricciarelli
Christian K. Riechmann
Grover Righter
Jean Risley
Noel F. Rivera-Silva
Andrew Roach
Clyde G. Roby Jr.
Ron Roehrig
Marco P. Roodzant
David Rorke
Alan Rosencrans
Seth Rosenthal
Vicki H. Rosenthal
David Rosenzweig
John C. Rowe
Craig Rubin
T. W. Rudolph
Edward M. Ruiz
Philip Rushton
William H. Rutter
Doris R. Ryan
Art Sabsevitz
Li San-Li
Steve Sarapata
Robert T. Sarr
Maude Sawyer
Ashok Saxena
Lorne H. Schachter
Ing. Helmut Schaefer
Stuart G. Schaefer
Mark A. Schaffer
Norman K. Scherer
Lee Schermerhorn
Curt F. Schimmel
Gerhard Schmitt
Eric R. Schott
David W. Schuler
Fritz Schulz
Stephen Schwarn
Glen Seeds
Warren T. Sekino
James W. Selkaitio
Karen Sheaffer
Dan Shia
Roy Shiderly
Del Shoemaker
Thomas E. Shonk
W. Olin
Mark Silverman
Peter P. Silvester
Leo Sintonen
Jacob Slonim
Donald F. Smith
Randy D. Smith
Thomas Smith

Jeff Smits
Richard Sniderman
James S. Soddy
Glenn R. Sogge
Steven E. Sommars
Richard M. Stallman
Bert Stanleigh
Lucy V. Stasiak
Dennis D. Steinauer
Daniel Steinberg
Harlan Stenn
Eric W. Stephenson
Armando P. Stettner
Douglas Steves
Robert G. Stewart
Kevin George Edward Sto
Steve Sutton
Tatsuo Suzuki
Robert Swartz
Dick Swee
Robert Switzer
Katalin Szenes
Theodore Tabloski
Darryl Taft
Ravi Tavakley
Colin B. Taylor
David R. Taylor
Marc J. Teller
Donn S. Terry
Jack Test
William J. Thomass
Hendrik-Jan Thomassen
Paul U. Thompson
Daniel F. Tiernan
Michael D. Tilson
Claus Tondering
Teoman Topcubasi
Woody Trant
Walter E. Tuvell
Andrew Twigger
David R. Uhrluab
Andrej Valencic
Peter van der Linden
J. van Katwijk
Joel Wagner
William Waite
R. Neil Walker
Raymond Walker
Michael Wallace
Richard Ward
Elizabeth Watson
John Waycott
Alan G. Weaver
Waldo M. Wedel
H. J. Weegenaar
Richard A. Weeks
Larry A. Wehr
Bruce Weiner
Brian Weis
Bob Weisbeck
Perry C. Weller
Alice Wellschlager
Andrew E. Wheeler
Gary L. Whisenhunt
Vicky White
Dietrich Wiegandt
John R. Willams
David Willcox
Judy Williams
Paul A. Willis
David Wilner
Ken Witte
Andrew Wolfe
David J. Woodend
John Wu
Shinichi Yamada
Ifen Yang
Margaret Yang
Shuitsu Yoshida
Nian Zu You
Peter M. Young
Peggy Younger
Hilary Zaloom
Steve Zanoni
John Carl Zeigler
Marvin Zelkowitz
Hubert Zimmerman
Harry Zint
Tatsuo Zuzuki

During the process of developing this standard, the Working Group sought to find problems with the standard in a manner that was both fun and which would publicize the standard. The contest is described in the Rationale (see **WeirdNIX** §B.1.2.12). Part of the prize was publication of the names of the winners.

Our special thanks to:

- Paul Gootherts (winner in *Most Serious* category).
- Michael Gersten (winner in *Most Demented* category).

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

Heinz Lycklama	*/usr/group Institutional Representative*
Michael Lambert	*X/Open Institutional Representative*
John S. Quarterman	*USENIX Institutional Representative*

Belton Allen
David F. Athersych
Karl Auerbach
A. L. Barrese
David Bernstein
Kabekode Bhat
Kathy Bohrer*
Robert Borochoff
Paul L. Borrill
James P. Bound
Phyllis Eve Bregman*
A. Winsor Brown
Luis-Felipe Cabrera
N. A. Camillone
James A. Capps
John Carmichael
Steven L. Carter
John Caywood
Chan Fung Chong
Anthony V. Cincotta*
Robert Claeson

Richard Clark
Clement T. Cole
Kenneth N. Cole
N. C. Comsudi
Peter Cook
Richard Cornelius
Bill Corwin
William T. Cox
Donald W. Cragun*
Dave Decot
William DeKeyser
G. C. Dimitriou
David L. Dodge
Terence S. Dowling
Stephen Dum*
Larry Dwyer*
John Earls
Michael A. Edmonds*
Ron Elliott
Philip H. Enslow Jr.
Fran Fadden
Kenneth T. Faubel*
Glenn S. Fields
Terence Fong
John Gertwagen
Kenneth R. Gibb
Randall W. Graves
John Gregg
William F. Griffeth Jr.
Robert C. Groman
Judy Guist
Gregory Guthrie
Douglas A. Gwyn
Charles E. Hammons
Carol Harkness
Katherine Harper
Dale Harris
Stephen M. Head
Myron Hect
Hans H. Heilborn
Thomas S. Heines
Jim Hightower
Lee A. Hollaar
Leon M. Holmes
Thomas F. Houghton
Randall Howard
Andrew R. Huber
James D. Isaak*
Richard James
Hal Jespersen
Cyndi Schmidt Johansen
Greg Jones
Michael Karels
Sol Kavy
Lorraine C. Kevra*
Jeffrey S. Kimmel
Dale L. Kirkland
Joshua W. Knight
David Korn
Donald Kretsch
D. Richard Kuhn
Takahiko Kuki
Tom M. Kurihara
Robin B. Lake
J. Eli Lamb
Mike G. Lambert
Doris Lebovits
Maggie Lee*
Robert Lenk*
David Lennert
Marjorie Levitz
Kin Li
Gottfried W. R. Luderer
Joseph F. P. Luhukay
Craig Lund
Heinz Lycklama
Roger Martin
Joberto S. B. Martins
Yoshihiro Matsumoto
Shane P. McCarron*
James McGinness
Marshall Kirk McKusick
Doug L. Michels
Gary Miller*
Jim Moe
James Mooney
Cliff Moore
Martha Nalebuff
Landon Curt Noll
Fred Noz
Alan F. Nugent
Perry Nuhn
Robin T. O'Neill
Charles Oestereicher
Jim R. Oldroyd
Mark Parenti
James A. Parker
S. Parthasarathy
Craig Partridge
John C. Penney
Sugar Peter
Jeffrey Picciotto
P. J. Plauger
Tom Plum
Gerald Powell*
Scott Preece
James Purtilo
John S. Quarterman
Jane Radatz
Jeffrey A. Ramsey
Wendy Rauch-Hindin
Carol L. Raye
Christopher J. Riddick
Grover Righter
Clyde G. Roby Jr.
Vicki H. Rosenthal
Doris R. Ryan
Art Sabsevitz
Robert T. Sarr
Ashok Saxena
Lorne H. Schachter
N. F. Schneidewind
Leonard W. Seagren
Glen Seeds
Mukesh Singhal
Leo Sintonen
Randy D. Smith
Jeff Smits*
Richard Sniderman
Steven E. Sommars
Richard M. Stallman
Daniel Steinberg*
Douglas Steves*
Dick Swee
Robert Switzer
Ravi Tavakley
Meng-Hee Teng
Donn S. Terry*
Graham Tigg
Gary F. Tom
Walter E. Tuvell
Andrew Twigger
Mark-Rene Uchida
L. David Umbaugh
Michael W. Vannier
M. B. Wagner
John Walz
Larry A. Wehr*
Bruce Weiner
Brian Weis
Peter J. Weyman
Gary Whisenhuut
David Willcox
John David Wu*
Oren Yuen
Janusz Zalewsk
Hilary Zaloom
Marvin Zelkowitz

In the preceding list, those individuals identified with asterisks (*) served during the balloting period as Technical Reviewers for resolving comments and objections to designated portions of the standard.

When the IEEE Standards Board approved this standard on August 22, 1988, it had the following membership:

Donald C. Fleckenstein, *Chairman* **Marco Migliaro,** *Vice Chairman*
Andrew G. Salem, *Secretary*

Arthur A. Blaisdell
Fletcher J. Buckley
James M. Daly
Stephen R. Dillon
Eugene P. Fogarty
Thomas L. Hannan
Kenneth D. Hendrix
Theodore W. Hissey, Jr
Jack M. Kinn
Frank D. Kirschner
Frank C. Kitzantides
Joseph L. Koepfinger*
Irving Kolodny
Edward Lohse
John E. May, Jr
Lawrence V. McCall
L. Bruce McClung
Richard E. Mosher
L. John Rankine
Gary S. Robinson
Frank L. Rose
Helen M. Wood
Karl H. Zaininger
Donald W. Zipse

*Member Emeritus

Contents

SECTION PAGE

1. Scope 21

2. Definitions and General Requirements 23
   2.1 Terminology. 23
   2.2 Conformance. 24
   2.3 General Terms. 28
   2.4 General Concepts. 35
   2.5 Error Numbers. 37
   2.6 Primitive System Data Types. 40
   2.7 Environment Description. 40
   2.8 C Language Definitions. 42
   2.9 Numerical Limits. 44
   2.10 Symbolic Constants. 47

3. Process Primitives 49
   3.1 Process Creation and Execution. 49
      3.1.1 Process Creation. 49
      3.1.2 Execute a File. 50
   3.2 Process Termination. 53
      3.2.1 Wait for Process Termination. 54
      3.2.2 Terminate a Process. 56
   3.3 Signals. 57
      3.3.1 Signal Concepts. 57
      3.3.2 Send a Signal to a Process. 62
      3.3.3 Manipulate Signal Sets. 63
      3.3.4 Examine and Change Signal Action. 64
      3.3.5 Examine and Change Blocked Signals. 66
      3.3.6 Examine Pending Signals. 67
      3.3.7 Wait for a Signal. 67
   3.4 Timer Operations. 68
      3.4.1 Schedule Alarm. 68
      3.4.2 Suspend Process Execution. 68
      3.4.3 Delay Process Execution. 69

4. Process Environment 71
   4.1 Process Identification. 71
      4.1.1 Get Process and Parent Process IDs. 71
   4.2 User Identification. 71
      4.2.1 Get Real User, Effective User, Real Group, and Effective Group IDs. 71
      4.2.2 Set User and Group IDs. 72
      4.2.3 Get Supplementary Group IDs. 73
      4.2.4 Get User Name. 74
   4.3 Process Groups. 75
      4.3.1 Get Process Group ID. 75
      4.3.2 Create Session and Set Process Group ID. 75

4.3.3 Set Process Group ID for Job Control. 75
4.4 System Identification. 76
4.4.1 System Name. 76
4.5 Time. 77
4.5.1 Get System Time. 77
4.5.2 Process Times. 78
4.6 Environment Variables. 79
4.6.1 Environment Access. 79
4.7 Terminal Identification. 79
4.7.1 Generate Terminal Pathname. 79
4.7.2 Determine Terminal Device Name. 80
4.8 Configurable System Variables. 80
4.8.1 Get Configurable System Variables. 80

5. Files and Directories 83
5.1 Directories. 83
5.1.1 Format of Directory Entries. 83
5.1.2 Directory Operations. 83
5.2 Working Directory. 85
5.2.1 Change Current Working Directory. 85
5.2.2 Working Directory Pathname. 86
5.3 General File Creation. 87
5.3.1 Open a File. 87
5.3.2 Create a New File or Rewrite an Existing One. 90
5.3.3 Set File Creation Mask. 90
5.3.4 Link to a File. 90
5.4 Special File Creation. 92
5.4.1 Make a Directory. 92
5.4.2 Make a FIFO Special File. 93
5.5 File Removal. 94
5.5.1 Remove Directory Entries. 94
5.5.2 Remove a Directory. 95
5.5.3 Rename a File. 96
5.6 File Characteristics. 97
5.6.1 File Characteristics: Header and Data Structure. 97
5.6.2 Get File Status. 99
5.6.3 File Accessibility. 100
5.6.4 Change File Modes. 101
5.6.5 Change Owner and Group of a File. 102
5.6.6 Set File Access and Modification Times. 103
5.7 Configurable Pathname Variables. 105
5.7.1 Get Configurable Pathname Variables. 105

6. Input and Output Primitives 109
6.1 Pipes. 109
6.1.1 Create an Inter-Process Channel. 109
6.2 File Descriptor Manipulation. 110
6.2.1 Duplicate an Open File Descriptor. 110

SECTION PAGE

6.3 File Descriptor Deassignment. 111
6.3.1 Close a File. 111
6.4 Input and Output. 111
6.4.1 Read from a File. 111
6.4.2 Write to a File. 113
6.5 Control Operations on Files. 115
6.5.1 Data Definitions for File Control Operations. 116
6.5.2 File Control. 117
6.5.3 Reposition Read/Write File Offset. 120

7. Device- and Class-Specific Functions 123
7.1 General Terminal Interface. 123
7.1.1 Interface Characteristics. 123
7.1.1.1 Opening a Terminal Device File. 123
7.1.1.2 Process Groups. 123
7.1.1.3 The Controlling Terminal. 124
7.1.1.4 Terminal Access Control. 124
7.1.1.5 Input Processing and Reading Data. 125
7.1.1.6 Canonical Mode Input Processing. 125
7.1.1.7 Non-Canonical Mode Input Processing. 126
7.1.1.8 Writing Data and Output Processing. 127
7.1.1.9 Special Characters. 127
7.1.1.10 Modem Disconnect. 128
7.1.1.11 Closing a Terminal Device File. 129
7.1.2 Settable Parameters. 129
7.1.2.1 *termios* Structure. 129
7.1.2.2 Input Modes. 129
7.1.2.3 Output Modes. 131
7.1.2.4 Control Modes. 131
7.1.2.5 Local Modes. 132
7.1.2.6 Special Control Characters. 133
7.1.2.7 Baud Rate Functions. 134
7.2 General Terminal Interface Control Functions. 136
7.2.1 Get and Set State. 136
7.2.2 Line Control Functions. 137
7.2.3 Get Foreground Process Group ID. 139
7.2.4 Set Foreground Process Group ID. 139

8. Language-Specific Services for the C Programming Language 141
8.1 Referenced C Language Routines. 141
8.1.1 Extensions to Time Functions. 142
8.1.2 Extensions to *setlocale*() Function. 144
8.2 FILE-Type C Language Functions. 145
8.2.1 Map a Stream Pointer to a File Descriptor. 145

SECTION PAGE

8.2.2 Open a Stream on a File Descriptor. 146
8.2.3 Interactions of Other FILE-Type C Functions. 146
8.2.4 Operations on Files — the *remove*() Function. 149
8.3 Other C Language Functions. 150
8.3.1 Non-Local Jumps. 150
8.3.2 Set Time Zone. 150

9. System Databases 151
9.1 System Databases. 151
9.2 Database Access. 151
9.2.1 Group Database Access. 151
9.2.2 User Database Access. 152

10. Data Interchange Format 155
10.1 Archive/Interchange File Format. 155
10.1.1 Extended `tar` Format. 155
10.1.2 Extended `cpio` Format. 159
10.1.3 Multiple Volumes. 162

APPENDICES

A. Related Standards 163
A.1 Related Standards—Open System Environment. 163
A.2 Standards Closely Related to the 1003.1 Document. 164
A.2.1 System Interface. 164
A.2.2 C Language Standard. 164
A.2.3 Shell and Utilities. 164
A.2.4 Verification Testing. 166
A.2.5 Realtime Extensions. 166
A.2.6 Ada Language Bindings. 166
A.2.7 Trusted System Extensions. 166
A.2.8 Open System Guidelines. 166
A.2.9 System Administration Extensions. 166
A.2.10 Networking Standards. 167
A.2.11 Language Standards. 167
A.2.12 Graphics Standards. 167
A.2.13 Database Standards. 168
A.3 Industry Open Systems Publications. 168
A.4 US Government Standards. 168
A.4.1 Federal Information Processing Standards (FIPS). 168
A.4.2 Trusted Systems. 168

B. Rationale and Notes 171
B.1 Introduction. 171
B.1.1 Scope. 172
B.1.2 Purpose. 173
B.1.3 Base Documents. 178
B.1.4 POSIX and the C Standard. 180

SECTION PAGE

B.1.5 Organization. 183
B.2 Definitions and General Requirements. 186
B.2.1 Terminology. 186
B.2.2 Conformance. 188
B.2.3 General Terms. 193
B.2.4 General Concepts. 203
B.2.5 Error Numbers. 206
B.2.6 Primitive System Data Types. 208
B.2.7 Environment Description. 210
B.2.8 C Language Definitions. 211
B.2.9 Numerical Limits. 213
B.2.10 Symbolic Constants. 216
B.3 Process Primitives. 216
B.3.1 Process Creation and Execution. 216
B.3.2 Process Termination. 219
B.3.3 Signals. 223
B.3.4 Timer Operations. 234
B.4 Process Environment. 235
B.4.1 Process Identification. 235
B.4.2 User Identification. 235
B.4.3 Process Groups. 236
B.4.4 System Identification. 237
B.4.5 Time. 238
B.4.6 Environment Variables. 239
B.4.7 Terminal Identification. 239
B.4.8 Configurable System Variables. 239
B.5 Files and Directories. 241
B.5.1 Directories. 241
B.5.2 Working Directory. 243
B.5.3 General File Creation. 244
B.5.4 Special File Creation. 245
B.5.5 File Removal. 245
B.5.6 File Characteristics. 246
B.5.7 Configurable Pathname Variables. 249
B.6 Input and Output Primitives. 250
B.6.1 Pipes. 251
B.6.2 File Descriptor Manipulation. 251
B.6.3 File Descriptor Deassignment. 252
B.6.4 Input and Output. 252
B.6.5 Control Operations on Files. 255
B.7 Device- and Class-Specific Functions. 258
B.7.1 General Terminal Interface. 260
B.7.2 General Terminal Interface Control Functions. 264
B.8 Language-Specific Services for the C Programming Language. 265
B.8.1 Referenced C Language Routines. 265
B.8.2 FILE-Type C Language Functions. 269
B.8.3 Other C Language Functions. 272

SECTION PAGE

B.9 System Databases. 272
B.9.1 System Databases. 272
B.9.2 Database Access. 273
B.10 Data Interchange Format. 273
B.10.1 Archive/Interchange File Format. 273
B.11 Bibliographic Notes. 279
B.11.1 Related Standards. 279
B.11.2 Historical Implementations. 279
B.11.3 Historical Application Programming Tutorials. 281

Identifier Index 283

Topical Index 287

LIST OF TABLES

Table 2-1. Primitive System Data Types 40

Table 2-2. Minimum Values 45

Table 2-3. Run-Time Increasable Values 45

Table 2-4. Run-Time Invariant Values (Possibly Indeterminate) 46

Table 2-5. Pathname Variable Values 46

Table 2-6. Symbolic Constants for the *access*() Function 47

Table 2-7. Symbolic Constants for the *lseek*() Function 47

Table 2-8. Compile-Time Symbolic Constants 48

Table 2-9. Execution-Time Symbolic Constants 48

Table 3-1. Required Signals 58

Table 3-2. Job Control Signals 58

Table 4-1. *uname*() Structure Members 77

Table 4-2. Configurable System Variables 81

Table 5-1. *stat* Structure 98

Table 5-2. Configurable Pathname Variables 105

Table 6-1. *cmd* Values for *fcntl*(). 115

Table 6-2. File Descriptor Flags Used For *fcntl*(). 116

Table 6-3. *l_type* Values For Record Locking With *fcntl*(). 116

Table 6-4. *oflag* Values For *open*(). 116

Table 6-5. File Status Flags Used For *open*() and *fcntl*(). 116

Table 6-6. File Access Modes Used For *open*() and *fcntl*(). 116

Table 6-7. Mask For Use With File Access Modes. 117
Table 6-8. *flock* Structure 118
Table 6-9. *fcntl()* Return Values 119
Table 7-1. *termios* Structure 129
Table 7-2. *termios c_iflag* Field 129
Table 7-3. *termios c_cflag* Field 131
Table 7-4. *termios c_lflag* Field 132
Table 7-5. *termios c_cc* Special Control Characters 134
Table 7-6. *termios* Baud Rate Values 135
Table 9-1. *group* Structure 152
Table 9-2. *passwd* Structure 152
Table 10-1. `tar` Header Block 156
Table 10-2. Byte-Oriented `cpio` Archive Entry 160
Table 10-3. Values for `cpio` *c_mode* Field 162
Table B-1. Typographical Conventions 184
Table B-2. Short Name Usages 185

Portable Operating System Interface for Computer Environments

1. Scope

This standard 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.

Initially, the focus of this standard will be 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 the standard into two parts—a section defining core requirements independent of any programming language, and a section composed of programming language bindings.

The core requirements section will define a set of required services common to any programming language that can be reasonably expected to form a language binding to this standard. 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 the standard. The other will contain a standardized interface for language-specific services. Any implementation claiming conformance to IEEE Std 1003.1-1988 with any language binding shall comply with both sections of the language binding.

This standard is comprised of 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 standard:

(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.

(See Appendix A for information about ongoing efforts in some of these areas.)

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

This standard has been defined exclusively at the source code level. The objective is that a Strictly Conforming POSIX Application source program can be translated to execute on a conforming implementation.

2. Definitions and General Requirements

2.1 Terminology. The following terms are used in this standard:

implementation-defined. A value or behavior is *implementation-defined* if the implementation defines and documents the requirements for correct program construct and correct data.

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

shall. In this standard, the word *shall* is to be interpreted as a requirement on the implementation or on *Strictly Conforming POSIX Applications*, where appropriate.

should. With respect to implementations, the word *should* is to be interpreted as an implementation recommendation, but not a requirement. With respect to applications, the word *should* is to be interpreted as recommended programming practice for applications and a requirement for *Strictly Conforming POSIX Applications*.

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

undefined. A value or behavior is *undefined* if the standard imposes no portability requirements on applications for erroneous program construct, erroneous data, or use of an indeterminate value. 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 **Conforming POSIX Application Using Extensions** §2.2.2.3.

unspecified. A value or behavior is *unspecified* if the standard imposes no portability requirements on applications for a correct program construct or correct data. 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 **Conforming POSIX Application Using Extensions** §2.2.2.3.

2.2 Conformance.

2.2.1 Implementation Conformance.

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

(1) The system shall support all required interfaces defined within this standard. These interfaces shall support the functional behavior described herein.

(2) The system may provide additional functions or facilities not required by this standard. 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 standard. In such cases, the system documentation 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 Application*.

2.2.1.2 Documentation. A document with the following information shall be available for an implementation claiming conformance to IEEE Std 1003.1-1988. This document shall have the same structure as this standard, with the information presented in the appropriately numbered sections. The document shall not contain information about extended facilities or capabilities outside the scope of this standard.

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

The document shall describe the contents of 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 document shall describe the behavior of the implementation for all *implementation-defined* features identified in this standard. The document is not required to describe those features identified as *undefined* or *unspecified*.

The document should specify the behavior of the implementation in those sections of this standard where it is stated that implementations may vary.

2.2.1.3 Conforming Implementation Options. The following symbolic constants, described in the sections indicated, reflect implementation options for this standard that could warrant requirement by *Conforming POSIX Applications*, or in specifications of conforming systems, or both:

{NGROUPS_MAX}
: Multiple groups option (in **Run-Time Increasable Values** §2.9.3)

{_POSIX_JOB_CONTROL}
: Job control option (in **Compile-Time Symbolic Constants** §2.10.3)

{_POSIX_CHOWN_RESTRICTED}
: Administrative/security option (in **Execution-Time Symbolic Constants** §2.10.4)

The remaining symbolic constants in **Compile-Time Symbolic Constants** §2.10.3 and **Execution-Time Symbolic Constants** §2.10.4 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 Applications* or in distinguishing between conforming implementations.

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

2.2.2 Application Conformance. All applications claiming conformance to this standard shall use only **Language-Dependent Services for the C Programming Language** §2.2.3, and shall fall within one of the following categories:

2.2.2.1 Strictly Conforming POSIX Application. A *Strictly Conforming POSIX Application* is an application that requires only the facilities described in this standard and the applicable language standards. Such an application shall accept any behavior described in this standard as *implementation-defined*, and for symbolic constants, shall accept any value in the range permitted by this standard. Such applications are permitted to adapt to the availability of facilities whose availability is indicated by the constants in **<limits.h>** §2.9 and **<unistd.h>** §2.10.

2.2.2.2 Conforming POSIX Application.

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

2.2.2.2.2 <National Body> Conforming POSIX Application. A *<National Body> Conforming POSIX Application* differs from an *ISO Conforming POSIX Application* in that it also may use specific standards of a single ISO 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.

2.2.2.3 Conforming POSIX Application Using Extensions. A *Conforming POSIX Application Using Extensions* is an application that differs from a *Conforming POSIX Application* only in that it uses non-standard facilities which are consistent with this standard. Such an application shall fully document its requirements for these extended facilities, in addition to the documentation required of a *Conforming POSIX Application*. A *Conforming POSIX Application Using Extensions* shall be either an *ISO Conforming POSIX Application Using Extensions* or a *<National Body> Conforming POSIX Application Using Extensions* (see sections §2.2.2.2.1 and §2.2.2.2.2).

2.2.3 Language-Dependent Services for the C Programming Language. When the C Standard (*ANSI/X3.159-198x Programming Language C Standard*) is ratified, parts of it will be referenced to describe requirements also mandated by IEEE Std 1003.1-1988. The sections of the C Standard referenced to describe requirements for this standard are specified in Chapter 8. Chapter 8 also sets forth additions and amplifications to the referenced sections of the C Standard. Any implementation claiming conformance to IEEE Std 1003.1-1988 with the C Language Binding shall provide the facilities referenced in Chapter 8, along with any additions and amplifications Chapter 8 requires.

Although IEEE Std 1003.1-1988 references parts of the C Standard to describe some of its own requirements, conformance to the C Standard is unnecessary for conformance to IEEE Std 1003.1-1988. Any C Language implementation providing the facilities stipulated in Chapter 8 may claim conformance—however, it shall clearly state that its C language does not conform to the C Standard.

2.2.3.1 Types of Conformance. Implementations claiming conformance to IEEE Std 1003.1-1988 with the C Language Binding shall claim one of two types of conformance—conformance to IEEE Std 1003.1-1988, C Language Binding (C Standard Language-Dependent System Support), or to IEEE Std 1003.1-1988, C Language Binding (Common Usage C Language-Dependent System Support).

2.2.3.2 C Standard Language-Dependent System Support. Implementors shall meet the requirements of Chapter 8 using for reference the C Standard. Implementors shall clearly document the version of the C Standard referenced in fulfilling the requirements of Chapter 8.

Until the C Standard is ratified, implementors shall reference the draft of that document dated 13 May 1988 (X3J11/88-002). Implementors seeking to claim conformance using the draft C Standard shall claim conformance to IEEE Std 1003.1-1988, C Language Binding (C Standard Language-Dependent System Support). Those implementors shall clearly document that the draft version of the C Standard referred to in implementing Chapter 8 was the draft dated 13 May 1988. Implementations using the draft C Standard prior to the formal ratification of that standard should change their implementations to reflect changes to the C Standard once it is ratified.

2.2.3.3 Common Usage C Language-Dependent System Support. Implementors, instead of referencing the C Standard, shall provide the routines and support required in Chapter 8 using common usage as guidance. Implementors shall meet all the requirements of Chapter 8 except where references are made to the C Standard. In places where the C Standard is referenced, implementors shall provide equivalent support in a manner consistent with common usage of the C programming language. Implementors shall document any differences between the interface provided and the interface that would have been provided had the C Standard been referenced instead of common usage. Implementors shall clearly document the version of the C Standard referenced in documenting interface differences and should issue updates on differences for all new versions of the C Standard.

Until the C Standard is ratified, implementors claiming conformance to IEEE Std 1003.1-1988, C Language Binding (Common Usage Language-Dependent System Support) shall reference the draft C Standard dated 13 May 1988 when documenting interface differences between their implementation of Chapter 8 and implementations of Chapter 8 based on the C Standard.

Where a function has been introduced by the C Standard, 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. If the function is not implemented, it shall be documented as a difference from the C Standard as required above.

2.2.4 Other C Language Related Specifications. The following rules apply to the usage of C language library functions; each of the statements in this section applies to the detailed function descriptions in Chapters 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 standard.

(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.

2.3 General Terms. The following are definitions of terms peculiar to this standard:

absolute pathname. *See* **pathname resolution** §2.4.

access mode. A form of access permitted to a *file*.

address space. The memory locations that can be referenced by a *process*.

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 standard that need special privileges. There may be zero or more such means.

background process group. Any *process group* that is a member of a *session* which has established a connection with a *controlling terminal* that is not in the *foreground process group*.

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.

C Standard. The abbreviated name for the *ANSI/X3.159-198x Programming Language C Standard*.

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

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 **General Terminal Interface** §7.1. Other *character special files* have no structure defined by this standard and their use is implementation-defined.

child process. *See process*.

clock tick. The number of intervals per second, defined by {CLK_TCK}, used to express the value in type *clock_t*.

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*.

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 **General Terminal Interface** §7.1) cause *signals* to be sent to all *processes* in the *process group* associated with the *controlling terminal*.

current working directory. *See working directory*.

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

directory. A *file* that contains *directory entries*. No two *directory entries* in the same *directory* shall have the same name.

directory entry (*or* **link**). An object that associates a *filename* with a *file*. Several *directory entries* can associate names with the same *file*.

dot. The *filename* consisting of a single dot character (`.`). *See* **pathname resolution** §2.4.

dot-dot. The *filename* consisting solely of two dot characters (`..`). *See* **pathname resolution** §2.4.

effective group ID. An attribute of a *process* that is used in determining various permissions, including **file access permissions** §2.4. *See group ID*. This value is subject to change during the *process lifetime*, as described in *setgid*() §4.2.2 and *exec* §3.1.2.

effective user ID. An attribute of a *process* that is used in determining various permissions, including **file access permissions** §2.4. *See user ID*. This value is subject to change during the *process lifetime*, as described in *setuid*() §4.2.2 and *exec* §3.1.2.

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

empty string (*or* **null string**). A character array whose first element is a null character.

Epoch. The time 0 hours, 0 minutes, 0 seconds, January 1, 1970 Coordinated Universal Time. *See seconds since the Epoch*.

feature test macro. A `#defined` symbol used to determine whether a particular set of features will be included from a header. *See* **Symbols From The C Standard** §2.8.1.

FIFO special file (*or* **FIFO**). A type of *file*. Data written to such a *file* is read on a first-in-first-out basis. Other characteristics of *FIFO*s are described under *open*() §5.3.1, *read*() §6.4.1, *write*() §6.4.2, and *lseek*() §6.5.3.

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.

file description. *See open file description*.

file descriptor. A per-*process* unique, non-negative integer used to identify an open *file* for the purpose of file access.

file group class. 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.

file mode. An object containing the *file permission bits* and other characteristics of a *file*, as described in **<sys/stat.h>** §5.6.1.

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** §2.4. A *filename* is sometimes referred to as a *pathname component*.

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*.

file other class. 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*.

file owner class. 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*.

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 **<sys/stat.h>** §5.6.1. The detailed usage of the *file permission bits* in access decisions is described in **file access permissions** §2.4.

file serial number. A per-*file system* unique identifier for a *file*. *File serial numbers* are unique throughout a *file system*.

file system. A collection of *files* and certain of their attributes. It provides a name space for *file serial numbers* referring to those *files*.

foreground process group. 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*. The *foreground process* group has certain privileges when accessing its *controlling terminal* that are denied to *background process groups*. *See* **Terminal Access Control** §7.1.1.4.

foreground process group ID. The *process group ID* of the *foreground process group*.

group ID. Each system user is a member of at least one group. A group is identified by a *group ID*, a non-negative integer that can be contained in an object of type *gid_t*. 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*.

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 **Symbolic Constants** §2.10).

link. *See directory entry*.

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

mode. A collection of attributes that specifies a *file*'s type and its access permissions. (*See* **file access permissions** §2.4).

null string. *See empty string*.

open file. A *file* that is currently associated with a *file descriptor*.

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 in **Data Definitions for File Control Operations** §6.5.1), and *file access modes* (Table 6-6) are attributes of an *open file description*.

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*.

parent directory. When discussing a *directory*, the *directory* containing the *directory entry* for the *directory* under discussion. 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*.

parent process. *See process*.

parent process ID. A new *process* 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*.

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

pathname. A string that is used to identify a *file*. It 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 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 under **pathname resolution** §2.4.

pathname component. *See filename*.

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** §2.4.

portable filename character set. For a *filename* to be portable across conforming implementations of IEEE Std 1003.1-1988, 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.

privilege. *See appropriate privileges*.

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*() as the *child process*.

process group. Each *process* in the *system* is a member of a *process group* that is identified by a *process group ID*. This grouping permits the signaling of related *processes*. A newly-created *process* joins the *process group* of its creator.

process group ID. Each *process group* in the *system* is uniquely identified during its lifetime by a positive integer that can be contained in a *pid_t* called a *process group ID*. A *process group ID* may not be reused by the *system* until the *process group lifetime* ends.

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

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, either due to *process* termination or calling the *setsid*() or *setpgid*() functions.

process ID. Each *process* in the *system* is uniquely identified during its lifetime by a positive integer that can be contained in a *pid_t* called a *process ID*. A *process ID* may 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* may 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.

process lifetime. 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.

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

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*. This value is subject to change during the *process lifetime*, as described in *setgid*() §4.2.2.

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*. This value is subject to change during the *process lifetime*, as described in *setuid*() §4.2.2.

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

relative pathname. *See* **pathname resolution** §2.4.

root directory. A *directory*, associated with a *process*, that is used in **pathname resolution** §2.4 for *pathnames* that begin with a *slash*.

saved set-group-ID. When the *saved set-user-ID* option is implemented, an attribute of a *process* that allows some flexibility in the assignment of the *effective group ID* attribute, as described in *setgid*() §4.2.2, and *exec* §3.1.2.

saved set-user-ID. When the *saved set-user-ID* option is implemented, an attribute of a *process* that allows some flexibility in the assignment of the *effective user ID* attribute, as described in *setuid*() §4.2.2, and *exec* §3.1.2.

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 year < 1970 or the value is negative, the relationship is undefined. If year ≥ 1970 and the value is non-negative, 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

session. 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 *setsid*() §4.3.2). Implementations which support *setpgid*() §4.3.3 can have multiple *process groups* in the same *session*.

session leader. A *process* that has created a *session* (see *setsid*() §4.3.2).

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

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.

slash. The literal character "/". This character is also known as *solidus* in ISO 8859/1.

supplementary group ID. A *process* has up to {NGROUPS_MAX} *supplementary group ID*s used in determining *file access permissions*, in addition to the *effective group ID*. The *supplementary group ID*s of a *process* are set to the *supplementary group ID*s 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 ID*s is unspecified.

system. An implementation of this standard.

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

terminal (*or* **terminal device).** A *character special file* that obeys the specifications of the **General Terminal Interface** §7.1.

user ID. Each system user is identified by a non-negative integer known as a *user ID* that can be contained in an object of type *uid_t*. 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*.

working directory (*or* **current working directory**). A *directory*, associated with a *process*, that is used in **pathname resolution** §2.4 for *pathnames* that do not begin with a *slash*.

2.4 General Concepts.

extended security controls. The access control (see *file access permissions*) and privilege (see **appropriate privileges** §2.3) 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 are described in this standard. These mechanisms shall not alter or override the defined semantics of any of the functions in this standard.

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*() §5.3.1, *creat*() §5.3.2, *mkdir*() §5.4.1, and *mkfifo*() §5.4.2, and are changed by *chmod*() §5.6.4. These bits are read by *stat*() or *fstat*() §5.6.2.

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 to which the process belongs, or if an alternate access control mechanism is enabled and it allows the requested access; otherwise access is denied.

file hierarchy. Files in the system are organized in a hierarchical structure in which all of the non-terminal 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*.

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.

file times update. Each file has three associated time values that are updated when file data has been accessed, file data has been modified, or file status has been changed, respectively. These values are returned in the file characteristics structure, as described in **<sys/stat.h>** §5.6.1.

For each function in this standard that reads or writes file data or changes the file status, the appropriate time-related fields are noted as *marked for update*. An implementation may update fields that are marked for update immediately, or 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*() §5.6.2 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.

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 *pathconf*() §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.

2.5 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 standard sets *errno* to zero to indicate an error.

If more than one error occurs in processing a function call, this standard 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 list, may generate errors included in this list under circumstances other than those described here, or may contain extensions or limitations that prevent some errors from occurring. The **Errors** subsection in each function description specifies which error conditions shall be required and which may be implementation-defined. Implementations shall not generate an error number different from the ones described here for error conditions described in this standard.

The following symbolic names identify the possible error numbers, in the context of functions specifically defined in this standard; these general descriptions are more precisely defined in the **Errors** sections of functions that return them. Only these symbolic names should be used in programs, since the actual value of the error number is implementation-defined. All values listed in this section shall be unique. The implementation-defined values for these names shall be found in the header **<errno.h>**.

[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; 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>** §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 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 no disk pack is 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; 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.

2.6 Primitive System Data Types. Some data types used by the various system functions are not defined as part of this standard, 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.
uid_t	Used for user IDs.

All of the types listed in Table 2-1 shall be arithmetic types; *pid_t* shall be a signed arithmetic type.

Additional implementation-defined type definitions may be given in this header. These definitions shall have names ending with *_t*. Such symbols do not require feature test macros to be visible when **<sys/types.h>** is included.

2.7 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>** §9.2.2).

LANG The name of the predefined setting for locale.

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 name of the user's login account, corresponding to the login name in the user database (see the description of the header **<pwd.h>**). The value shall be composed of characters from the **portable filename character set** §2.3.

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 non-zero-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 **Extensions to Time Functions** §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 *exec* §3.1.2).

2.8 C Language Definitions.

2.8.1 Symbols From The C Standard. Certain terms and symbols used in this standard are considered to be defined by the C programming language. The following terms are defined in the C Standard: *CLK_TCK*, *NULL*, *byte*, *character array*, *clock_t*, *header*, *null character*, *string*, *time_t*.

The term **NULL** *pointer* in this standard is equivalent to the term *null pointer* used in the C Standard. The symbol **NULL** shall be declared in **<unistd.h>**, with the same value as required by the C Standard, in addition to the several locations already required by the C Standard.

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. These reservations apply to this standard as well. Additionally, the C Standard 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 standard.

2.8.2 POSIX Symbols. Certain symbols in this standard are defined in headers. Some of those headers could also define other symbols than those defined by this standard, potentially conflicting with symbols used by the application. Also, this standard defines symbols which 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 standard, and other standards may define additional feature test macros. Feature test macros shall be defined in the compilation of an application before a `#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 members to a structure or union without controlling the visibility of those members with a feature test macro.

The following feature test macro is defined:

Name	Description
_POSIX_SOURCE	The program expects that the symbols defined by this standard will be provided by the environment. Where extensions are permitted in a header, but no explicit constraint on the form of the name is provided by this standard, the extensions shall not be made visible by this feature test macro.

The exact meaning of feature test macros depends on the type of C language support chosen:

2.8.2.1 C Standard Language-Dependent Support. If there are no feature test macros present in a program, only the set of symbols defined by the C Standard shall be present. For each feature test macro present, only the symbols specified by that feature test macro plus those of the C Standard shall be defined when a header is included.

2.8.2.2 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 standard is implementation-defined.

If _POSIX_SOURCE is defined before any header is included, no symbols other than those from the C Standard and those made visible by feature test macros defined for the program (including _POSIX_SOURCE) will be visible.

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

2.8.3 Headers and Function Prototypes. Implementations claiming C Standard 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*.

These function prototypes (if required) shall appear in the headers listed below. If a function is not listed below, it shall have its prototype appear in **<unistd.h>**, which is presumed to be `#include`-ed whenever any function declared in it is used, whether or not it is mentioned in the **Synopsis** section for that function. Functions also described in the C Standard (see **Language-Specific Services for the C Programming Language** §8) shall have their prototypes appear in the headers defined for them in the C Standard. The requirements about visibility of symbols in **POSIX Symbols** §2.8.2 shall be honored.

<dirent.h> *opendir*() §5.1.2, *readdir*() §5.1.2, *rewinddir*() §5.1.2, *closedir*() §5.1.2.

<fcntl.h>	*open*() §5.3.1, *creat*() §5.3.2, *fcntl*() §6.5.2.
<grp.h>	*getgrgid*() §9.2.1, *getgrnam*() §9.2.1.
<pwd.h>	*getpwuid*() §9.2.2, *getpwnam*() §9.2.2.
<setjmp.h>	*sigsetjmp*() §8.3.1, *siglongjmp*() §8.3.1.
<signal.h>	*kill*() §3.3.2, *sigemptyset*() §3.3.3, *sigfillset*() §3.3.3, *sigaddset*() §3.3.3, *sigdelset*() §3.3.3, *sigismember*() §3.3.3, *sigaction*() §3.3.4, *sigprocmask*() §3.3.5, *sigpending*() §3.3.6, *sigsuspend*() §3.3.7.
<stdio.h>	*fileno*() §8.2.1, *fdopen*() §8.2.2.
<sys/stat.h>	*umask*() §5.3.3, *mkdir*() §5.4.1, *mkfifo*() §5.4.2, *stat*() §5.6.2, *fstat*() §5.6.2, *chmod*() §5.6.4.
<sys/times.h>	*times*() §4.5.2.
<sys/utsname.h>	*uname*() §4.4.1.
<sys/wait.h>	*wait*() §3.2.1, *waitpid*() §3.2.1.
<termios.h>	*cfgetospeed*() §7.1.2.7, *cfsetospeed*() §7.1.2.7, *cfgetispeed*() §7.1.2.7, *cfsetispeed*() §7.1.2.7, *tcgetattr*() §7.2.1, *tcsetattr*() §7.2.1, *tcsendbreak*() §7.2.2, *tcdrain*() §7.2.2, *tcflush*() §7.2.2, *tcflow*() §7.2.2.
<time.h>	*time*() §4.5.1, *tzset*() §8.3.2.
<utime.h>	*utime*() §5.6.6.

2.9 Numerical Limits. The following subsections list magnitude limitations imposed by a specific implementation. The braces notation, {LIMIT}, is used in the standard to indicate these values, but the braces are not part of the name.

2.9.1 C Language Limits. Certain limits used in this standard are considered to be defined in the C programming language. The following limits are defined in the C Standard (for information on that standard, see the section on **C Language Standard** §A.2.2): {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}.

2.9.2 Minimum Values. The symbols in Table 2-2 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 standard. Related symbols are defined elsewhere in this standard which reflect the actual implementation and which may 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.

2.9.3 Run-Time Increasable Values. The magnitude limitations in Table 2-3 shall be fixed by specific implementations.

A **Strictly Conforming POSIX Application** shall assume that the value supplied by **<limits.h>** in a specific implementation is the minimum that pertains whenever the **Strictly Conforming POSIX Application** is run under that implementation. 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

Table 2-2. 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

Table 2-3. Run-Time Increasable Values

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

sysconf() §4.8.1 function.

2.9.4 Run-Time Invariant Values (Possibly Indeterminate). A definition of one of the values in Table 2-4 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*() §4.8.1 function.

2.9.5 Pathname Variable Values. The values in Table 2-5 may be constants within an implementation, or may vary from one pathname to another.

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

Table 2-4. 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}

Table 2-5. 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 **Canonical Mode Input Processing** §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}

A definition of one of the values from 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 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*( ) §5.7.1 function.

2.10 Symbolic Constants. A conforming implementation shall have the header **<unistd.h>**. This header defines the symbolic constants and structures referenced elsewhere in the standard. The constants defined by this header are shown below. The actual values of the constants are implementation-defined.

2.10.1 Symbolic Constants for the *access*() Function. The constants used by the *access*() function are shown in Table 2-6.

Table 2-6. 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.

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.

2.10.2 Symbolic Constant for the *lseek*() Function. The constants used by the *lseek*() function are shown in Table 2-7.

Table 2-7. 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*.

2.10.3 Compile-Time Symbolic Constants for Portability Specifications. The constants in Table 2-8 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.

Although a Strictly Conforming POSIX 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 *sysconf*() §4.8.1.

Table 2-8. 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 198808L. This value will change with each published version or revision of this standard to indicate the (4-digit) year and (2-digit) month that the standard was approved by the IEEE Standards Board.

2.10.4 Execution-Time Symbolic Constants for Portability Specifications. The constants in Table 2-9 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 standard as *implementation-defined*.

If any of the constants in Table 2-9 are not defined in the header **<unistd.h>**, the value varies depending on the file to which it is applied. See *pathconf*() §5.7.1.

If any of the constants in Table 2-9 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.

Table 2-9. Execution-Time Symbolic Constants

Name	Description
{_POSIX_CHOWN_RESTRICTED}	The use of the *chown*() §5.6.5 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 **<termios.h>** §7.1.2 can be disabled using this character value, if it is defined. See *tcgetattr*() and *tcsetattr*() §7.2.1.

All of the constants in Table 2-9, whether defined in **<unistd.h>** or not, may be queried with respect to a specific file using the *pathconf*() or *fpathconf*() functions.

3. Process Primitives

The functions described in this chapter perform the most primitive operating system services dealing with processes, interprocess signals, and timers. All attributes of a process that are specified in this standard 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 ( )
```

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 **Directory Operations** §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 *times*() §4.5.2).

(6) File locks previously set by the parent are not inherited by the child. (See *fcntl*() §6.5.2.)

(7) Pending alarms are cleared for the child process. (See *alarm*() §3.4.1.)

(8) The set of signals pending for the child process is initialized to the empty set. (See **<signal.h>** §3.3.1.)

All other process characteristics defined by this standard shall be the same in the parent and the child processes. The inheritance of process characteristics

not defined by this standard is implementation-defined.

3.1.1.3 Returns. Upon successful completion, *fork*() shall return to the child process a value of zero and shall return to the parent process the process ID of the child process, and 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 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 (path, arg0, arg1, ..., argn, (char *) 0)
char *path, *arg0, *arg1, ..., *argn;

int execv (path, argv)
char *path, *argv[ ];

int execle (path, arg0, arg1, ..., argn, (char *) 0, envp)
char *path, *arg0, *arg1, ..., *argn, *envp[ ];

int execve (path, argv, envp);
char *path, *argv[ ], *envp[ ];

int execlp (file, arg0, arg1, ..., argn, (char *) 0)
char *file, *arg0, *arg1, ..., *argn;

int execvp (file, argv)
char *file, *argv[ ];

extern char **environ;
```

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 (*argc*, *argv*)
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 does not contain a slash character, the path prefix for this file is obtained by a search of the directories passed as the environment variable **PATH** (see **Environment Description** §2.7). If this environment variable is not present, the results of the search are implementation-defined.

The arguments *arg0*, *arg1*, ..., *argn* are pointers to null-terminated character strings. These strings constitute the argument list available to the new process image. The list is terminated by a **NULL** pointer. The argument *arg0* should point to a filename that is associated with the process being started by one of the *exec* functions.

The argument *argv* is an array of character pointers to null-terminated strings. The 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 argument *envp* is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process image. The *envp* array is terminated by a **NULL** pointer.

For those forms not containing an *envp* pointer (*execl*(), *execv*(), *execlp*(), and *execvp*()) the environment for the new process image 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 **Documentation** §2.2.1.2) 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 *fcntl*() §6.5.2, **<fcntl.h>** §6.5.1). For those file descriptors that remain open, all attributes of the open file description, including file locks (see *fcntl*() §6.5.2), remain unchanged by this function call.

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 **<signal.h>** §3.3.1).

If the set-user-ID mode bit of the new process image file is set (see *chmod*() §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 *alarm*() §3.4.1)
(9) current working directory
(10) root directory
(11) file mode creation mask (see *umask*() §5.3.3)
(12) process signal mask (see *sigprocmask*() §3.3.5)
(13) pending signals (see *sigpending*() §3.3.6)
(14) *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* (see *times*() §4.5.2)

All process attributes defined by this standard and not specified in this section shall be the same in the new and old process images. The inheritance of process attributes not defined by this standard is implementation-defined.

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.

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 new process image's argument list and environment list is greater than the system-imposed limit of {ARG_MAX} bytes.

[EACCES] Search permission is denied for a directory listed in the new process image file's path prefix, or the new process image file denies execution permission, or the new process image file is not a regular file and the implementation does not support execution of files of its type.

[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 new process image file's pathname do not exist, or the *path* or *file* argument points to an empty string.

[ENOTDIR] A component of the new process image file's path prefix 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 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, **Environment Description** §2.7.

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 some signals are received (see **<signal.h>** §3.3.1).

The *exit*() and *abort*() functions shall be as described in the C Standard (see **C Language Standard** §A.2.2). Both *exit*() and *abort*() shall terminate a process with the consequences specified in *_exit*() §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(stat_loc)
int *stat_loc;

pid_t waitpid (pid, stat_loc, options)
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 delivery of a signal whose action is either to execute a signal-catching function or to terminate the process. 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*)
: Evaluates to a non-zero value if status was returned for a child process that terminated normally.

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

WIFSIGNALED(*stat_val*)
: Evaluates to a non-zero value if status was returned for a child process that terminated due to the receipt of a signal that was not caught (see **<signal.h>** §3.3.1).

WTERMSIG(*stat_val*)
: If the value of WIFSIGNALED(*stat_val*) is non-zero, this macro evaluates to the number of the signal that caused the termination of the child process.

WIFSTOPPED(*stat_val*)
: Evaluates to a non-zero value if status was returned for a child process that is currently stopped.

WSTOPSIG(*stat_val*)
: If the value of WIFSTOPPED(*stat_val*) is non-zero, this macro evaluates to the number of the signal that caused the child process to stop.

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*), and WIFSTOPPED(**stat_loc*) shall evaluate to a non-zero 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*) and WIFSIGNALED(**stat_loc*) shall evaluate to a non-zero 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.

If a parent process terminates without waiting for all of its child processes to terminate, the remaining child processes shall be assigned a new parent process

ID corresponding to an implementation-defined system process.

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*, it has at least one child process specified by *pid* for which status is not available, and status is not available for any process specified by *pid*, 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 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 (status)
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 calling process's termination and the low order 8 bits of *status* are made available to it; see *wait* §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 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 *sigsetops* §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 standard and supported by the implementation shall have distinct, positive integral values. The value zero is reserved for use as the null signal (see *kill*() §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 standard. 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 *kill*() §3.3.2 and *sigaction*() §3.3.4.

Table 3-1. Required Signals

Symbolic Constant	Default Action	Description
SIGABRT	1	Abnormal termination signal, such as is initiated by the *abort*() function (as defined in the C Standard).
SIGALRM	1	Timeout signal, such as initiated by the *alarm*() function (see *alarm*() §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 **Modem Disconnect** §7.1.1.10) or death of controlling process (see *_exit*() §3.2.2).
SIGILL	1	Detection of an invalid hardware instruction.
SIGINT	1	Interactive attention signal (see **Special Characters** §7.1.1.9).
SIGKILL	1	Termination signal (cannot be caught or ignored).
SIGPIPE	1	Write on a pipe with no readers (see *write*() §6.4.2).
SIGQUIT	1	Interactive termination signal (see **Special Characters** §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.

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 **Special Characters** §7.1.1.9).
SIGTTIN	3	Read from control terminal attempted by a member of a background process group (see **Terminal Access Control** §7.1.1.4).
SIGTTOU	3	Write to control terminal attempted by a member of a background process group (see **Terminal Access Control** §7.1.1.4).

Default actions for Tables 3-1 and 3-2 are as follows:

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

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 **Signal Actions** §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 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 standard under which the implementation generates signals. (See **Documentation** §2.2.1.2.)

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 *exec* §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 standard are specified in the preceding tables.

(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 *sigaction*() §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 as 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.

(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* (*signo*)
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.

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

All IEEE Std 1003.1-1988 functions not in the above table and all functions defined in the C Standard not stated to be callable from a signal-catching function are considered to be *unsafe* with respect to signals. If any function that is *unsafe* is interrupted by a signal-catching function that then calls any function that is *unsafe*, the behavior is undefined.

3.3.1.4 Signal Effects on Other Functions. Signals affect the behavior of certain functions defined by this standard 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, sig)
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 **<signal.h>** §3.3.1 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 a receiving process's effective user ID has been altered through use of the S_ISUID mode bit (see **<sys/stat.h>** §5.6.1), the implementation may still permit the application to receive a signal sent by the parent process or by a process with the same real 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 implementation-defined 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 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 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 (set)
sigset_t *set;

int sigfillset (set)
sigset_t *set;

int sigaddset (set, signo)
sigset_t *set;
int signo;

int sigdelset (set, signo)
sigset_t *set;
int signo;

int sigismember (set, signo)
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 **<signal.h>** §3.3.1).

The *sigemptyset*() function initializes the signal set pointed to by the argument *set*, such that all signals defined in this standard are excluded.

The *sigfillset*() function initializes the signal set pointed to by the argument *set*, such that all signals defined in this standard 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 and delete the individual signal specified by the value of the argument *signo* 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 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 (sig, act, oact)
int sig;
struct sigaction *act, *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 **<signal.h>** §3.3.1.

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.

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 process's signal mask 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, 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.

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 **<signal.h>** §3.3.1.

3.3.4.5 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 (how, set, oset)
int how;
sigset_t *set, *oset;
```

3.3.5.2 Description. The *sigprocmask*() function is used to examine or change (or both) the calling process's signal mask. 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>** §3.3.1:

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 process's signal mask 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.

If the *sigprocmask*() function fails, the process's signal mask 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 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 (set)
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 standard does not specify any error conditions that are required to be detected for the *sigpending*() function. Some errors may be detected under implementation-defined conditions.

3.3.6.5 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 (sigmask)
sigset_t *sigmask;
```

3.3.7.2 Description. The *sigsuspend*() function replaces the process's signal mask 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 **<signal.h>** §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 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 (seconds)
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 to not actually 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. The *alarm*() function shall return the amount of time remaining in seconds before the system is scheduled to generate the SIGALRM signal, or zero if there is no previous *alarm*() request.

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 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 ( )
```

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 References. *alarm*() §3.4.1, *kill*() §3.3.2, *wait* §3.2.1, **Signal Effects on Other Functions** §3.3.1.4.

3.4.3 Delay Process Execution.

Function: *sleep*()

3.4.3.1 Synopsis.

unsigned int sleep (*seconds*)
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 if the SIGALRM signal is being ignored or blocked from delivery, it is unspecified whether *sleep*() returns when the SIGALRM signal is scheduled. If the signal is being blocked, it is also unspecified whether it remains pending after the *sleep*() function returns or it 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 examines or changes either the time a SIGALRM is scheduled to be generated, the action associated with the SIGALRM signal, or whether the SIGALRM signal is blocked from delivery, the results are unspecified.

If a signal-catching function interrupts 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 *sigsetjmp*() §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 References. *alarm*() §3.4.1, *pause*() §3.4.2, *sigaction*() §3.3.4.

4. Process Environment

4.1 Process Identification.

4.1.1 Get Process and Parent Process IDs.

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

4.1.1.1 Synopsis.

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

pid_t getpid ( )

pid_t getppid ( )
```

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 **Description**.

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 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 ( )

uid_t geteuid ( )

gid_t getgid ( )

gid_t getegid ( )
```

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 **Description**.

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 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)
uid_t uid;

int setgid (gid)
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 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 (gidsetsize, grouplist)
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 is returned. The values of array entries with indices larger than or equal to the returned value are undefined.

It is unspecified whether the effective group ID of the calling process is included in or omitted from the returned list of supplementary group IDs.

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 References. *setgid*() §4.2.2.

4.2.4 Get User Name.

Functions: *getlogin*(), *cuserid*()

4.2.4.1 Synopsis.

```
char *getlogin ( )

#include <stdio.h>

char *cuserid (s)
char *s;
```

4.2.4.2 Description. These functions return a string giving a name of the user associated with the current process. The *cuserid*() function returns a name associated with the effective user ID of the process, and the *getlogin*() function returns the name associated by the login activity with the control terminal.

The recommended procedure is either to call the *cuserid*() function, or to call *getlogin*() and, if it fails, to call the *getpwuid*() function with the value returned by the *getuid*() function.

The *getlogin*() function returns a pointer to the user's login name. The same user ID may be shared by several login names. Therefore, to ensure that the correct user database entry is found, the *getlogin*() function should be used with the *getpwnam*() function.

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

The *cuserid*() function generates a character representation of the login name of the owner of the current process. If *s* is not a **NULL** pointer, it is assumed that *s* points to an array of at least L_cuserid bytes; the representation is returned in this array. The symbolic constant L_cuserid is defined in **<stdio.h>**, and shall have a value greater than zero.

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

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

If *s* is a **NULL** pointer, the result from *cuserid*() is generated in an area that may be static, the address of which is returned. If the login name cannot be found, *cuserid*() returns a **NULL** pointer. If *s* is not a **NULL** pointer, *s* is returned. If the login name cannot be found, the null character shall be placed at **s*. The return value from *cuserid*() may point to static data and therefore may be overwritten by each call.

The implementation of the *cuserid*() function may use the *getpwnam*() function, so the results of a user's call to either routine may be overwritten by a subsequent call to the other routine.

4.2.4.4 Errors. This standard does not specify any error conditions that are required to be detected for the *cuserid*() or *getlogin*() functions. Some errors may be detected under implementation-defined conditions.

4.2.4.5 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 ( )
```

4.3.1.2 Description. The *getpgrp*() function returns the process group ID of the calling process.

4.3.1.3 Returns. See **Description**.

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 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 ( )
```

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.

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 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, pgid)
pid_t pid, 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 References. *getpgrp*() §4.3.1, *setsid*() §4.3.2, *tcsetpgrp*() §7.2.4, *exec* §3.1.2.

4.4 System Identification.

4.4.1 System Name.

Function: *uname*()

4.4.1.1 Synopsis.

```
#include <sys/utsname.h>

int uname (name)
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 that the system is running on.

Each of these data items is a null-terminated character array.

The format of each member is implementation-defined. The system documentation (see **Documentation** §2.2.1.2) 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 non-negative value is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

4.4.1.4 Errors. This standard does not specify any error conditions that are required to be detected for the *uname*() function. Some errors may be detected under implementation-defined conditions.

4.5 Time.

4.5.1 Get System Time.

Function: *time*()

4.5.1.1 Synopsis.

```
#include <time.h>

time_t time (tloc)
time_t *tloc;
```

4.5.1.2 Description. The *time*() function returns the value of time in **seconds since the Epoch** §2.3.

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 standard does not specify any error conditions that are required to be detected for the *time*() function. Some errors may be detected under implementation-defined conditions.

4.5.2 Process Times.

Function: *times*()

4.5.2.1 Synopsis.

#include <sys/times.h>

clock_t times (*buffer*)
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 in {CLK_TCK}ths of a second.

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 *wait* §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 {CLK_TCK}ths of a second, 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 standard does not specify any error conditions that are required to be detected for the *times*() function. Some errors may be detected under implementation-defined conditions.

4.5.2.5 References. *exec* §3.1.2, *fork*() §3.1.1, *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** (*name*)
char *__*name*__**;**

4.6.1.2 Description. The *getenv*() function searches the environment list (see **Environment Description** §2.7) 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 standard does not specify any error conditions that are required to be detected for the *getenv*() function. Some errors may be detected under implementation-defined conditions.

4.6.1.5 References. *environ* §3.1.2, **Environment Description** §2.7.

4.7 Terminal Identification.

4.7.1 Generate Terminal Pathname.

Function: *ctermid*()

4.7.1.1 Synopsis.

#include <stdio.h>

char *__**ctermid** (*s*)
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 a character array 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, of if the function is unsuccessful.

4.7.1.4 Errors. This standard does not specify any error conditions that are required to be detected for the *ctermid*() function. Some errors may be detected under implementation-defined conditions.

4.7.1.5 References. *ttyname*() §4.7.2.

4.7.2 Determine Terminal Device Name.

Functions: *ttyname*(), *isatty*()

4.7.2.1 Synopsis.

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

int isatty (fildes)
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 standard does not specify any error conditions that are required to be detected for the *ttyname*() or *isatty*() functions. Some errors may be detected under implementation-defined conditions.

4.8 Configurable System Variables.

4.8.1 Get Configurable System Variables.

Function: *sysconf*()

4.8.1.1 Synopsis.

```
#include <unistd.h>

long sysconf (name)
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>** §2.9 or **<unistd.h>** §2.10 (or **<time.h>** from the C Standard for {CLK_TCK}), 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}
{CLK_TCK}	{_SC_CLK_TCK}
{NGROUPS_MAX}	{_SC_NGROUPS_MAX}
{OPEN_MAX}	{_SC_OPEN_MAX}
{_POSIX_JOB_CONTROL}	{_SC_JOB_CONTROL}
{_POSIX_SAVED_IDS}	{_SC_SAVED_IDS}
{_POSIX_VERSION}	{_SC_VERSION}

The value of {CLK_TCK} is permitted to be evaluated at run-time by the C Standard (and thus by this standard). The value returned by *sysconf*() for {_SC_CLK_TCK} shall be the same as that returned by {CLK_TCK}.

4.8.1.3 Returns. If *name* is an invalid value, *sysconf*() shall return −1. If the variable corresponding to *name* is not defined on 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>** §2.9 or **<unistd.h>** §2.10. 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.

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 **Input and Output Primitives** §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 implementation-defined.

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 character array *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 (dirname)
char *dirname;

struct dirent *readdir (dirp)
DIR *dirp;

void rewinddir (dirp)
DIR *dirp;

int closedir (dirp)
DIR *dirp;
```

5.1.2.2 Description. The type *DIR*, which is defined in the header **<dirent.h>** §5.1.1, 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 section. 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 *open*() §5.3.1. If a file descriptor is used, the FD_CLOEXEC flag shall be set on that file descriptor; see **<fcntl.h>** §6.5.1.

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. If entries for dot or dot-dot exist, one entry shall be returned for dot and one entry shall be returned for dot-dot; otherwise they shall not be returned.

The pointer returned by *readdir*() points to data which may be overwritten by another call to *readdir*() on the same directory stream. This data 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 not changed 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 any component of *dirname* or read permission is denied for *dirname*.

[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.

[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 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 (path)
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 References. *getcwd*() §5.2.2.

5.2.2 Working Directory Pathname.

Function: *getcwd*()

5.2.2.1 Synopsis.

char ***getcwd** (*buf*, *size*)
char **buf*;
int *size*;

5.2.2.2 Description. The *getcwd*() function copies an absolute pathname of the current working directory to the character array pointed to by the argument *buf* and returns a pointer to the result. The *size* argument is the size in bytes of the character array pointed to by the *buf* argument. If *buf* is a **NULL** pointer, the behavior of *getcwd*() is 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 is 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 less than or equal to zero.

[ERANGE] The *size* argument is greater than zero, but is 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 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 (path, oflag, ...)
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 which 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 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 **<fcntl.h>** §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 process's effective user ID; 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 process's effective group ID. The file permission bits (see **<sys/stat.h>** §5.6.1) shall be set to the value of *mode* except those set in the process's file mode creation mask (see *umask*() §5.3.3). When bits in *mode* other than the file permission bits are set, the effect is implementation-defined. 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 implementation-defined.

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 **The Controlling Terminal** §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 directories. 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 non-negative 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 which 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 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, **Signal Effects on Other Functions** §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 (path, mode)
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 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 (cmask)
mode_t cmask;
```

5.3.3.2 Description. The *umask*() routine sets the process's file mode creation mask to *cmask* and returns the previous value of the mask. Only the file permission bits (see **<sys/stat.h>** §5.6.1) of *cmask* are used; the meaning of the other bits is implementation-defined.

The process's file mode creation mask 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 previous value of the file mode creation mask is returned.

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 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 (path1, path2)
char *path1, *path2;
```

5.3.4.2 Description. The argument *path1* points to a pathname naming an existing file. The argument *path2* 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 *path1* 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 *path2* exists.

[EMLINK] The number of links to the file named by *path1* would exceed {LINK_MAX}.

[ENAMETOOLONG] The length of the *path1* or *path2* 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 *path1* does not exist; or either *path1* or *path2* 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 *path1* 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 *path2* and the file named by *path1* are on different file systems and the implementation does not support links between file systems.

5.3.4.5 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 (path, mode)
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 process's file creation mask (see *umask*() §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 directory's owner ID is set to the process's effective user ID. 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 process's effective group ID.

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 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 (path, mode)
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 process's file creation mask (see *umask*() §5.3.3). When bits in *mode* other than the file permission bits are set, the effect is implementation-defined.

The FIFO's owner ID shall be set to the process's effective user ID. The FIFO's group ID shall be set to the group ID of the directory in which the FIFO is being created or to the process's effective group ID.

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] A component of the path prefix denies search permission.
[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 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 (*path*)
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 file's link count 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 file's link count 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 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 (*path*)
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 directory is the root directory or the current working directory of any process, the effect of this function is implementation-defined.

If the directory's link count 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. A return value of zero indicates success. A return value of −1 indicates that an error has occurred and an error code has been stored in *errno*.

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 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 non-existent 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 References. *mkdir*() §5.4.1, *unlink*() §5.5.1.

5.5.3 Rename a File.

Function: *rename*()

5.5.3.1 Synopsis.

int rename (*old*, *new*)
char **old*;
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 file's link count 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. A return value of zero indicates success. A return value of −1 indicates that an error has occurred and an error code has been stored in *errno*.

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.

[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 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*().

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 standard. The value of the member *st_nlink* shall be set to the number of links to the file.

Table 5-1. *stat* Structure

Member Type	Member Name	Description
mode_t	*st_mode*	File mode (see **<sys/stat.h> File Modes** §5.6.1.2).
ino_t	*st_ino*	File serial number.
dev_t	*st_dev*	ID of device containing this file. File serial number and device ID taken together uniquely identify the file within the system.
nlink_t	*st_nlink*	Number of links.
uid_t	*st_uid*	User ID of the file's owner.
gid_t	*st_gid*	Group ID of the file's group.
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.

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 non-zero value if the test is true, zero if the test is false.

S_ISDIR(*m*) Test macro for directory file.
S_ISCHR(*m*) Test macro for character special file.
S_ISBLK(*m*) Test macro for block special file.
S_ISREG(*m*) Test macro for regular file.
S_ISFIFO(*m*) Test macro for pipe or 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 process's effective user ID 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 file's group 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 standard. 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 by **file times update** §2.4.

All the functions in this standard that change these fields directly describe those changes in the context of the functions' definitions. Other functions that directly change *st_atime*, *st_mtime*, or *st_ctime* shall be implementation-defined.

Times are given in **seconds since the Epoch** §2.3.

5.6.1.4 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).

5.6.2 Get File Status.

Functions: *stat*(), *fstat*()

5.6.2.1 Synopsis.

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

int stat (path, buf)
char *path;
struct stat *buf;

int fstat (fildes, buf)
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 **file times update** §2.4 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>** §5.6.1, 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 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 File Accessibility.

Function: *access*()

5.6.3.1 Synopsis.

```
#include <unistd.h>

int access (path, amode)
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 **Symbolic Constants for the *access*() Function** *§2.10.1* for the description of these symbolic constants.

If any access permission is to be checked, each shall be checked individually, as described in **file access permissions** §2.4. 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 References. *chmod*() §5.6.4, *stat*() §5.6.2, **<unistd.h>** §2.10.

5.6.4 Change File Modes.

Function: *chmod*()

5.6.4.1 Synopsis.

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

int chmod (path, mode)
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 **<sys/stat.h>** §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 **file access permissions** §2.4. 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, and if the group ID of the file does not match the effective group ID or one of the supplementary group IDs, and if the file is a regular file, bit S_ISGID (set group ID on execution) in the file's mode shall be cleared upon successful return from *chmod*().

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 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 (path, owner, group)
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 with an effective user ID equal to the user ID of the file, but without appropriate privileges, if and only if *owner* is equal to the file's user ID and *group* is equal either to the calling

process's effective group ID 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 **<sys/stat.h>** §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 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 (path, times)
char *path;
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** §2.3.

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** and the calling process's effective user ID has write access to the file but does not match the owner of the file and the calling process does not have the appropriate privileges.

[EROFS] The *named* file resides on a read-only file system.

5.6.6.5 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 (path, name)
char *path;
int name;

long fpathconf (fildes, name)
int fildes, 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>** §2.9 or **<unistd.h>** §2.10 and the symbolic constants, defined in **<unistd.h>**, that are the corresponding values used for *name*.

Table 5-2. Configurable Pathname Variables

Variable	*name* Value	Notes
{LINK_MAX}	{_PC_LINK_MAX}	1
{MAX_CANON}	{_PC_MAX_CANON}	2
{MAX_INPUT}	{_PC_MAX_INPUT}	2
{NAME_MAX}	{_PC_NAME_MAX}	3, 4
{PATH_MAX}	{_PC_PATH_MAX}	4, 5
{PIPE_BUF}	{_PC_PIPE_BUF}	6
{_POSIX_CHOWN_RESTRICTED}	{_PC_CHOWN_RESTRICTED}	7
{_POSIX_NO_TRUNC}	{_PC_NO_TRUNC}	3, 4
{_POSIX_VDISABLE}	{_PC_VDISABLE}	2

The following Notes apply to the entries in Table 5-2:

1. If *path* or *fildes* refers to a directory, the value returned applies to the directory itself.
2. The behavior is undefined if *path* or *fildes* does not refer to a terminal file.
3. If *path* or *fildes* refers to a directory, the value returned applies to the filenames within the directory.
4. The behavior is undefined if *path* or *fildes* does not refer to a directory.
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 *filedes* refers to a pipe or 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* refer to any other type of file, the behavior is undefined.
7. If *path* or *fildes* refer to a directory, the value returned applies to any files defined in this standard, other than directories, that exist or can be created within the directory.

5.7.1.3 Returns. If *name* is an invalid value, the *pathconf*() and *fpathconf*() functions shall return –1.

If the variable corresponding to *name* has no limit for the path or file descriptor, the *pathconf*() and *fpathconf*() functions shall return –1 without changing *errno*.

If the implementation needs to use *path* to determine the value of *name* and the implementation does not support the association of *name* with the file specified by *path*, or if the process did not have the appropriate privileges to query the file specified by *path*, or *path* does not exist, the *pathconf*() function shall return –1.

If the implementation needs to use *fildes* to determine the value of *name* and the implementation does not support the association of *name* with the file specified by *fildes*, or if *fildes* is an invalid file descriptor, the *fpathconf*() function shall return –1.

Otherwise, the *pathconf*() and *fpathconf*() functions return the current variable value for the file or directory without changing *errno*. The value returned shall not be more restrictive than the corresponding value described to the application when it was compiled with the implementation's **<limits.h>** §2.9 or **<unistd.h>** §2.10.

5.7.1.4 Errors. If any of the following conditions occur, the *pathconf*() and *fpathconf*() functions shall return –1 and set *errno* to the corresponding value:

[EINVAL] The value of *name* is invalid.

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

[EACCES] Search permission is denied for a component of the path prefix.

[EINVAL] The implementation does not support an association of the variable name with the specified file.

[ENAMETOOLONG] The length of the *path* argument exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.

[ENOENT] The named file does not exist or the *path* argument points to an empty string.

[ENOTDIR] A component of the path prefix is not a directory.

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

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

[EINVAL] The implementation does not support an association of the variable name with the specified file.

6. Input and Output Primitives

The functions in this chapter deal with input and output from files and pipes. Functions are also specified which 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 (*fildes*)
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 flag shall be clear on both file descriptors. (The *fcntl*() function can be used to set the O_NONBLOCK flag.)

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 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 (*fildes*)
int *fildes*;

int dup2 (*fildes*, *fildes2*)
int *fildes*, *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 less than zero or greater than {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*.

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*() and *dup2*() functions shall return –1 and set *errno* to the corresponding value:

[EBADF] The argument *fildes* is not a valid open file descriptor or *fildes2* is out of range.

[EMFILE] The number of file descriptors would exceed {OPEN_MAX}, or no file descriptors above *fildes2* are available.

6.2.1.5 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 (*fildes*)
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 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, **Signal Effects on Other Functions** §3.3.1.4.

6.4 Input and Output.

6.4.1 Read from a File.

Function: *read*()

6.4.1.1 Synopsis.

int read (*fildes*, *buf*, *nbyte*)
int *fildes*;
char **buf*;
unsigned *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 {INT_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 opened the pipe for writing.

When attempting to read a file (other than a pipe or FIFO) that supports non-blocking reads and has no data currently available:

(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.

(3) 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, 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 for implementation-defined reasons.

6.4.1.5 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, **Signal Effects on Other Functions** §3.3.1.4, terminal **Interface Characteristics** §7.1.1.

6.4.2 Write to a File.

Function: *write*()

6.4.2.1 Synopsis.

```
int write (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned 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, the *write*() function shall return zero and have no other results if the file is a regular file; otherwise, the results are implementation-defined.

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.

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 non-zero 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 {INT_MAX}, the result is implementation-defined.

Write requests to a pipe (or FIFO) shall be handled the same as 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: the *write*() function shall not block the process; write requests for {PIPE_BUF} or fewer bytes shall either succeed completely and return *nbyte*, or return −1 and set *errno* to [EAGAIN]; a *write*() request for greater than {PIPE_BUF} bytes shall either transfer what it can and return the number of bytes written, or transfer no data and return −1 with *errno* set to [EAGAIN]. Also, if a *write*() request is greater than {PIPE_BUF} bytes and all data previously written to the pipe has been read, *write*() shall transfer at least {PIPE_BUF} bytes.

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, 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 for implementation-defined reasons.

[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 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, **Signal Effects on Other Functions** §3.3.1.4.

6.5 Control Operations on Files.

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 doesn't exist.
O_EXCL	Exclusive use flag.
O_NOCTTY	Don't 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.

6.5.1 Data Definitions for File Control Operations. The header **<fcntl.h>** defines the following *requests* and *arguments* for the *fcntl*() §6.5.2 and *open*() §5.3.1 functions. The values within each of the Tables 6-1 through 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.

Table 6-7. Mask For Use With File Access Modes.

Constant	Description
O_ACCMODE	Mask for file access modes.

6.5.2 File Control.

Function: *fcntl*()

6.5.2.1 Synopsis.

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

int fcntl (fildes, cmd, ...)
int fildes, cmd;
```

6.5.2.2 Description. The function *fcntl*() provides for control over open files. The argument *fildes* is a file descriptor.

The available values for *cmd* are defined in the header **<fcntl.h>** §6.5.1, which shall include:

F_DUPFD Return a new file descriptor which 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 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 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, 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>** §6.5.1. 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, 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*. The file access mode shall not be changed by this function call. If any other bits are set in *arg*, the result is implementation-defined.

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 which 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>** §6.5.1 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 process's signal handler, *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>** §6.5.1 header, describes an advisory lock. It includes the members shown in Table 6-8.

Table 6-8. *flock* Structure

Member Type	Member Name	Description
short	*l_type*	F_RDLCK, F_WRLCK, or F_UNLCK.
short	*l_whence*	Flag for starting offset.
off_t	*l_start*	Relative offset in bytes.
off_t	*l_len*	Size; if 0 then until EOF.
pid_t	*l_pid*	Process ID of the process holding the lock, returned with F_GETLK.

When a shared lock has been set on a segment of a file, other processes shall be able to set shared locks on that segment or a portion of it. A shared lock prevents any other process from setting an exclusive lock on any portion of the protected area. A request for a shared lock shall fail if the file descriptor was not opened with read access.

An exclusive lock shall prevent any other process from setting a shared lock or an exclusive lock on any portion of the protected area. A request for an exclusive lock shall fail if the file descriptor was not opened with write access.

The value of *l_whence* is SEEK_SET, SEEK_CUR, or SEEK_END to indicate that the relative offset, *l_start* bytes, will be measured from the start of the file, current position, or end of the file, respectively. The value of *l_len* is the number of consecutive bytes to be locked. If *l_len* is negative, the result is implementation-defined. The *l_pid* field is only used with F_GETLK to return the process ID of the process holding a blocking lock. After a successful F_GETLK request, the value of *l_whence* shall be SEEK_SET.

Locks may start and extend beyond the current end of a file, but shall not start or extend before the beginning of the file. A lock shall be set to extend to the largest possible value of the file offset for that file if *l_len* is set to zero. If the *flock struct* has *l_whence* and *l_start* that point to the beginning of the file, and *l_len* of zero, the entire file shall be locked.

The calling process shall have only one type of lock set for each byte in the file. 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. 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 another process's locked region. 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.

Table 6-9. *fcntl*() Return Values

Request	Return Value
F_DUPFD	A new file descriptor.
F_GETFD	Value of the flags defined in Table 6-2, but the return value shall not be negative.
F_SETFD	Value other than −1.
F_GETFL	Value of file status flags and access modes, but the return value shall not be negative.
F_SETFL	Value other than −1.
F_GETLK	Value other than −1.
F_SETLK	Value other than −1.
F_SETLKW	Value other than −1.

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

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

[EACCES] or [EAGAIN]
The argument *cmd* is F_SETLK, the type of lock (*l_type*) is a shared lock (F_RDLCK) or exclusive lock (F_WRLCK), and the segment of a file to be locked is already exclusive-locked by another process, or the type is an exclusive lock and some portion of the segment of a file to be locked is already shared-locked or exclusive-locked by another process.

[EBADF] The *fildes* argument is not a valid file descriptor.
The argument *cmd* is F_SETLK or F_SETLKW, the type of lock (*l_type*) is a shared lock (F_RDLCK), and *fildes* is not a valid file descriptor open for reading.
The argument *cmd* is F_SETLK or F_SETLKW, the type of lock (*l_type*) is an exclusive lock (F_WRLCK), and *fildes* is not a valid file descriptor open for writing.

[EINTR] The argument *cmd* is F_SETLKW and the function was interrupted by a signal.

[EINVAL] The argument *cmd* is F_DUPFD and the third argument is negative or greater than or equal to {OPEN_MAX}.
The argument *cmd* is F_GETLK, F_SETLK, or F_SETLKW and the data *arg* points to 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 References. *close*() §6.3.1, *exec* §3.1.2, *open*() §5.3.1, **<fcntl.h>** §6.5.1, **Signal Effects on Other Functions** §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 (fildes, offset, whence)
int fildes, whence;
off_t offset;
```

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, SEEK_END are defined in the header **<unistd.h>** §2.10.

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 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>** §2.10.

7. Device- and Class-Specific Functions

7.1 General Terminal Interface. This section describes a general terminal interface that shall be provided to control asynchronous communications ports. It is implementation-defined whether this interface supports network connections or synchronous ports or both. Certain functions in this chapter apply only to the controlling terminal of a process. Where this is the case it will be 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 an application's standard input, output, and error files.

As described in *open*() §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 **Control Modes** §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 **Special Characters** §7.1.1.9.

If the implementation supports job control (if {_POSIX_JOB_CONTROL} is defined; see **Symbolic Constants** §2.10), 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. A terminal's foreground process group may be set or examined by a process, assuming the permission requirements in this section are met; see *tcgetpgrp*() §7.2.3 and *tcsetpgrp*() §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 **Terminal Access Control** §7.1.1.4.

---

* The P1003.2 Working Group is working on a definition and description of command interpreters. See **Shell and Utilities** §A.2.3.

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 *open*() §5.3.1), it is implementation-defined whether the terminal becomes the controlling terminal of the session leader. If a process which 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. A process relinquishes its controlling terminal when it creates a new session with the *setsid*() function, or when all file descriptors associated with the controlling terminal have been closed.

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 **Input Processing and Reading Data** §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 **Signal Names** §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 **Writing Data and Output Processing** §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 **Control Functions** §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 non-canonical mode. These modes are described in **Canonical Mode Input Processing** §7.1.1.6 and **Non-Canonical Mode Input Processing** §7.1.1.7. Additionally, input characters are processed according to the *c_iflag* (see **Input Modes** §7.1.2.2) and *c_lflag* (see **Local Modes** §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 non-canonical 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 non-canonical. The following sections, **Canonical Mode Input Processing** §7.1.1.6 and **Non-Canonical Mode Input Processing** §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 **Special Characters** §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 **Pathname Variable Values** §2.9.5.

Erase and kill processing occur when either of two special characters, the ERASE and KILL characters (see **Special Characters** §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 un-delimited 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 Non-Canonical Mode Input Processing. In non-canonical 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 bursty and short term data transmissions. If MIN is greater than {MAX_INPUT}, the response to the request is implementation-defined. 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 inter-byte timer and is activated after the first byte is received. Since it is an inter-byte 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 inter-byte timer is started. If MIN bytes are received before the inter-byte 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 a signal is received.

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 inter-byte 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.

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 **Output Modes** §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 *write*() §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 is recognized if the ISIG flag (see **Local Modes** §7.1.2.5) is enabled. Generates a SIGINT signal which 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 is recognized if the ISIG flag is enabled. Generates a SIGQUIT signal which 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 is recognized if the ICANON flag is set. Erases the last character in the current line; see **Canonical Mode Input Processing** §7.1.1.6. It 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 is recognized if the ICANON flag is set. 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 is 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 is recognized if the ICANON flag is set. It is the line delimiter ('\n').

EOL — Special character on input and is recognized if the ICANON flag is set. Is an additional line delimiter, like NL.

SUSP — If job control is supported (see **Special Control Characters** §7.1.2.6), the SUSP special character is recognized on input. 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 is recognized if the IXON (output control) or IXOFF (input control) flag is set. 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 is 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 is recognized if the ICANON flag is set; it is the '\r'. 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 **Special Control Characters** §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 interpreted as normal characters or as the special characters defined in this section.

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 **Control Modes** §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 *_exit*() §3.2.2. Any subsequent read from the terminal device returns with an end-of-file indication until the device is closed. Thus, processes that read a terminal file and test for end-of-file can terminate appropriately after a disconnect. 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 Settable Parameters.

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.

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

The types *tcflag_t* and *cc_t* shall be defined in the header **<termios.h>**. They shall be unsigned integral types.

The total size of the *termios* structure is implementation-defined.

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 and 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 **Control Modes** §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 number of bytes in the input queue from exceeding {MAX_INPUT}, 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. For example, at 110 baud, two stop bits

* CSIZE has historically described "character" size.

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 *open*() §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 it is directly connected to.

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 **Canonical Mode Input Processing** §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 expired between bytes. The time value represents tenths of seconds. See the **Non-Canonical Mode Input Processing** §7.1.1.7 section 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 non-canonical modes are shown in Table 7-5. The subscript name symbols in this table are defined in **<termios.h>**.

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 number of elements in the *c_cc* array, NCCS, is implementation-defined.

Implementations which do not support changing the START and STOP characters may ignore the character values in the *c_cc* array indexed by the VSTART

Table 7-5. *termios c_cc* Special Control Characters

Subscript Usage		
Canonical Mode	Non-Canonical 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

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 **Special Characters** §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 Functions.

Functions: *cfgetispeed*(), *cfgetospeed*(), *cfsetispeed*(), *cfsetospeed*()

7.1.2.7.1 Synopsis.

```
#include <termios.h>

speed_t cfgetospeed (termios_p)
struct termios *termios_p;

int cfsetospeed (termios_p, speed)
struct termios *termios_p;
speed_t speed;

speed_t cfgetispeed (termios_p)
struct termios *termios_p;

int cfsetispeed (termios_p, speed)
struct termios *termios_p;
speed_t speed;
```

7.1.2.7.2 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.

The input and output baud rates are stored in the *termios* structure. The values shown in Table 7-6 are supported. The name symbols in this table are defined in **<termios.h>**.

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

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.

cfgetospeed() returns the output baud rate stored in the *termios* structure pointed to by *termios_p*.

cfsetospeed() sets the output baud rate stored in the *termios* structure pointed to by *termios_p* to *speed*. The zero baud rate, B0, is used to terminate the connection. If B0 is specified, the modem control lines shall no longer be asserted. Normally, this will disconnect the line.

cfgetispeed() returns the input baud rate stored in the *termios* structure.

cfsetispeed() sets the input baud rate stored in the *termios* structure to *speed*. If the input baud rate is set to zero, the input baud rate will be specified by the value of the output baud rate. Both *cfsetispeed*() and *cfsetospeed*() return a value of zero if successful and −1 to indicate an error. Attempts to set unsupported baud rates shall be ignored, and it is implementation-defined whether an error is returned by any or all of *cfsetispeed*(), *cfsetospeed*(), or *tcsetattr*(). This refers both to changes to baud rates not supported by the hardware, and to changes setting the input and output baud rates to different values if the hardware does not support this.

7.1.2.7.3 Returns. See **Description**.

7.1.2.7.4 Errors. This standard 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 implementation-defined conditions.

7.1.2.7.5 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 section. 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, and 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 (fildes, termios_p)
int fildes;
struct termios *termios_p;

int tcsetattr (fildes, optional_actions, termios_p)
int fildes, optional_actions;
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.

The *tcsetattr*() function shall set the parameters associated with the terminal (unless support is required from the underlying hardware that is not available) from the *termios* structure referenced by *termios_p* as follows:

(1) If *optional_actions* is TCSANOW, the change shall occur immediately.

(2) If *optional_actions* is TCSADRAIN, the change shall occur after all output written to *fildes* has been transmitted. This function should be used when changing parameters that affect output.

(3) If *optional_actions* is TCSAFLUSH, the change shall occur after all output written to the object referred to by *fildes* has been transmitted, and all input that has been received but not read shall be discarded before the change is made.

The symbolic constants for the values of *optional_actions* are defined in **<termios.h>**.

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.

[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 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 (fildes, duration)
int fildes;
int duration;

int tcdrain (fildes)
int fildes;

int tcflush (fildes, queue_selector)
int fildes;
int queue_selector;

int tcflow (fildes, action)
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.

The *tcflush*() function shall discard 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 **Input Modes** §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 **Input Modes** §7.1.2.2.)

The symbolic constants for the values of *queue_selector* and *action* are defined in **<termios.h>**.

The default on open 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 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 (fildes)
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. 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 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 (fildes, pgrp_id)
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 a value not supported by the implementation.

[ENOSYS] The *tcsetpgrp*() function is not supported in this implementation.

[ENOTTY] The calling process does not have a controlling terminal, or the file is not the controlling terminal, or the controlling terminal is no longer associated with the session of the calling process.

[EPERM] The value of *pgrp_id* is a value supported by the implementation but does not match the process group ID of a process in the same session as the calling process.

8. Language-Specific Services for the C Programming Language

8.1 Referenced C Language Routines. The functions listed below will be described in the indicated sections of the C Standard. IEEE Std 1003.1-1988 with the C Language Binding comprises these functions, the extensions to them described in this chapter, and the rest of the requirements stipulated in this standard. The functions appended with plus signs (+) have requirements beyond those set forth in the C Standard. Any implementation claiming conformance to IEEE Std 1003.1-1988 with the C Language Binding shall comply with the requirements outlined in this chapter, the requirements stipulated in the rest of this standard, and the requirements in the indicated sections of the C Standard.

For requirements concerning conformance to this chapter, see **Language-Dependent Services for the C Programming Language** §2.2.3 and its subsections.

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 the IEEE Std 1003.1-1988 shall make no distinction between the "text streams" and the "binary streams" described in the C Standard.

For the *fseek*() function, if the specified position is beyond end-of-file, the consequences described in *lseek*() (see *lseek*() §6.5.3) shall occur.

The EXIT_SUCCESS macro, as used by the *exit*() function, shall evaluate to a value of zero.

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** §2.3, 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 **Environment Variables** §2.7) 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* Three or more 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 (**:**), digits, comma (**,**), minus (–), plus (**+**), and ASCII NUL are allowed.

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. Out of range values may cause unpredictable 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* ($1 \le n \le 365$). 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 ($0 \le n \le 365$). Leap days shall be counted, and it is possible to refer to February 29.

M*m*.*n*.*d* The d^{th} day ($0 \le d \le 6$) of week *n* of month *m* of the year ($1 \le n \le 5$, $1 \le m \le 12$, 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*() §8.3.2 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 (category, locale)
int category;
char *locale;
```

8.1.2.2 Description. The C Standard allows the specification of an implementation-defined native environment for the *setlocale*() function, which shall set a specific category to an implementation-defined default. For IEEE Std 1003.1-1988 systems, this corresponds to the value of the environment variables.

A specific category may be set to an implementation-defined default value by passing the *locale* argument with a pointer to a null string.

Possible values for *category* include:

LC_CTYPE
LC_COLLATE
LC_TIME
LC_NUMERIC
LC_MONETARY
Implementation-defined additional categories

In all cases, *setlocale*() shall first check the value of the corresponding environment variable (for example, LC_CTYPE for the LC_CTYPE category) and if valid (i.e., points to the name of a valid locale), *setlocale*() shall set the specified category of the international environment to that value and return the string corresponding to the locale set (i.e., the value of the environment variable, not `""`). If the value is invalid, *setlocale*() shall return a **NULL** pointer and the international environment is not changed by this function call.

If the environment variable corresponding to the specified category is not set or is set to the empty string, the behavior of *setlocale*() is implementation-defined, unless the **LANG** environment variable is set and valid in which case *setlocale*() will set the category to the corresponding value of **LANG**. In some implementations, this may default to a system-wide value, others may default to the `"C"` locale. Setting all categories to the implementation-defined default is similar to the previous usage, but it interrogates all the environment variables to determine the specific value to set. To set all categories in the international environment, *setlocale*() is invoked in the following manner:

```
setlocale(LC_ALL, "");
```

To satisfy this request, *setlocale*() first checks all the environment variables. If any environment variable is invalid, *setlocale*() returns a null pointer and the international environment is not changed by this function call. If all the relevant environment variables are valid, *setlocale*() sets the international environment to reflect the values of the environment variables. The categories are set in the following order:

LC_CTYPE
LC_COLLATE
LC_TIME
LC_NUMERIC
LC_MONETARY
Implementation-defined additional categories

Using this scheme, the categories corresponding to the environment variables will override the value of the **LANG** environment variable for a particular category.

If the **LANG** environment variable is not set or is set to the empty string, the behavior of

```
setlocale(category, "")
```

is implementation-defined.

8.2 FILE-Type C Language Functions. This section describes functions which make reference to the *FILE* type, (as described in the C Standard), and their interactions with other functions defined by this standard.

The terms *file position indicator* and *stream* are those defined by the C Standard.

A stream is considered local to a single process. After a *fork*() call each of the parent and child have distinct streams which 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 (stream)
FILE *stream;
```

8.2.1.2 Description. The *fileno*() function returns the integer file descriptor associated with the *stream* (see *open*() §5.3.1).

The following symbolic values in **<unistd.h>** §2.10 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

8.2.1.3 Returns. See **Description**. If an error occurs, a value of −1 is returned and *errno* is set to indicate the error.

8.2.1.4 Errors. This standard does not specify any error conditions that are required to be detected for the *fileno*() function. Some errors may be detected under implementation-defined conditions.

8.2.1.5 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 (fildes, type)
int fildes;
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 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 *type* of the stream must be 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.

fdopen() may cause the *st_atime* field of the underlying file to be marked for update.

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 standard does not specify any error conditions that are required to be detected for the *fdopen*() function. Some errors may be detected under implementation-defined conditions.

8.2.2.5 References. *open*() §5.3.1, *fopen*() (C Standard).

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 which 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 standard, 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 which is a stream is considered to be closed when either an *fclose*() or *freopen*() is executed on it (the result of *freopen*() is a new stream 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 which 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.

(1) For the first handle, the first applicable condition below shall apply. After the actions required below are taken, if the handle is still open, it may be closed.

(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 which is unbuffered, no action need be taken.

(d) If it is a stream which is line-buffered, and the last operation had the same effect on the underlying file as a *fgets*() or *fputs*(), no action need be taken. In the case of an *fgets*(), the effect above is to be interpreted as if no readahead that the implementation may choose to do had actually occurred.

(e) If it is a stream which is open for writing or append, (but not also open for reading) either a *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 a *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 *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 *_exit*().

(4) The *exec* functions shall be considered to make inaccessible all streams which 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, and all data is seen on reading sequentially) when the rules above are followed, regardless of the sequence of handles used. If the rules above are not followed, the result is unspecified.

(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 sections below, additional requirements on the standard I/O routines, beyond those in the C Standard, are given.

8.2.3.1 ***fopen*().** *fopen*() shall allocate a file descriptor as *open*() does. *fopen*() may cause the *st_atime* field of the underlying file to be marked for update.

The underlying function is *open*().

8.2.3.2 ***fclose*().** *fclose*() 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*().

If the file is not already at EOF, and the file is one capable of seeking, the file offset of the underlying open file description shall be adjusted so that the next operation on the open file description deals with the byte after the last one read from or written to the stream being closed.

8.2.3.3 ***freopen*().** *freopen*() has the properties of both *fclose*() and *fopen*().

8.2.3.4 ***fflush*().** *fflush*() 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 *read*(), *write*(), and *lseek*().

If the stream is open for reading, any unread data buffered in the stream shall be invalidated.

For a stream open for reading, if the file is not already at EOF, and the file is one capable of seeking, the file offset of the underlying open file description shall be adjusted so that the next operation on the open file description deals

with the byte after the last one read from or written to the stream being closed.

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*(), *vprintf*(), *vfprintf*(). 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*().

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*(). This 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 the completion of *fflush*() or *fclose*() on *stderr*, or *exit*() or *abort*().

8.2.3.9 *tmpfile*(). *tmpfile*() shall allocate a file descriptor as *fopen*() does.

8.2.3.10 *ftell*(). The underlying function is *fseek*(). The result of *ftell*() after an *fflush*() shall be the same as the result before the *fflush*().

8.2.3.11 Error Reporting. If any of the functions above return an error indication caused by a condition that would be detected by the corresponding underlying functions listed above, the value returned in *errno* shall be the one provided 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 program termination (see the C Standard for the conditions where program termination will not occur.)

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 Non-Local Jumps.

Functions: *sigsetjmp*(), *siglongjmp*()

8.3.1.1 Synopsis.

```
#include <setjmp.h>

int sigsetjmp (env, savemask)
sigjmp_buf env;
int savemask;

void siglongjmp (env, val)
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. If the value of the *savemask* argument is not zero, the *sigsetjmp*() function shall also save the process's current signal mask (see **<signal.h>** §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. If and only if the *env* argument was initialized by a call to the *sigsetjmp*() function with a non-zero *savemask* argument, the *siglongjmp*() function shall restore the saved signal mask.

8.3.1.3 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 ( )
```

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 **Extensions to Time Functions** §8.1.1.

9. System Databases

9.1 System Databases. The routines described in this section allow an application to access the two system databases that are described below.

The *group* database contains the following information for each group:

(1) group name
(2) numerical group ID
(3) list of the names or numbers of all users allowed in the group

The *user* database contains the following information for each user:

(1) login 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 implementation-defined.

9.2 Database Access.

9.2.1 Group Database Access.

Functions: *getgrgid*( ), *getgrnam*( )

9.2.1.1 Synopsis.

```
#include <grp.h>

struct group *getgrgid (gid)
gid_t gid;

struct group *getgrnam (name)
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 standard does not specify any error conditions that are required to be detected for the *getgrgid*() or *getgrnam*() functions. Some errors may be detected under implementation-defined conditions.

9.2.1.5 References. *getlogin*() §4.2.4.

9.2.2 User Database Access.

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

9.2.2.1 Synopsis.

```
#include <pwd.h>

struct passwd *getpwuid (uid)
uid_t uid;

struct passwd *getpwnam (name)
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's login 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.

The implementation of the *cuserid*() §4.2.4 function may use the *getpwnam*() function; thus the results of a user's call to either routine may be overwritten by a subsequent call to the other routine.

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 standard does not specify any error conditions that are required to be detected for the *getpwuid*() or *getpwnam*() functions. Some errors may be detected under implementation-defined conditions.

9.2.2.5 References. *cuserid*() §4.2.4, *getlogin*() §4.2.4.

10. Data Interchange Format

10.1 Archive/Interchange File Format. A conforming system shall provide a mechanism to copy files from a medium to the file hierarchy and copy files from the file hierarchy to a medium using the interchange formats described here. This standard 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*() §5.3.2 would when given the *mode* argument matching the file permissions supplied by the *mode* field of extended `tar` format or the *c_mode* field of the extended `cpio` format. A process with appropriate privileges shall restore the ownership and permissions exactly as recorded on the medium, except that the symbolic user and group IDs are used for the `tar` format, as described in **Extended tar Format** §10.1.1.

The *format-creating utility* is used to translate from the file system to the formats defined in this section, in an implementation-defined way, and the *format-reading utility* is used to translate from the formats defined in this section to a file system.

The headers of these formats are defined to use characters represented in ASCII, however, no restrictions are made about 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 ASCII.

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 ½-inch 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 1003.2 Working Group is working on this mechanism. See **Shell and Utilities** §A.2.3.

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 American Standard Code for Information Interchange, ASCII. For maximum portability between implementations, names should be selected from characters represented by the **portable filename character set** §2.3 as 8-bit characters with zero parity. If an extended character set beyond the portable character set is used, and the format-reading and format-creating utilities on the two distinct systems use the same extended character set, the file name shall be preserved. 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 standard; see *open*() §5.3.1, *stat*() §5.6.2, *chdir*() §5.2.1, *fcntl*() §6.5.2, and *opendir*() §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 non-null 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 in ASCII. Each numeric field is terminated by one of 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, a slash character and filename as the suffix. If the *prefix* contains non-null characters, *prefix*, a slash character, and *name* are concatenated without modification or addition of new characters to produce a new pathname. 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 99 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 standard. 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 file's owner and group, 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. If the *typeflag* field is set to CHARTYPE, BLKTYPE, or FIFOTYPE the meaning of the *size* field is implementation-defined and no data blocks are stored on the medium. 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 ASCII representation of the octal value of the modification time obtained from the *stat*() function.

The *chksum* field is the ASCII 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.

ASCII digit '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 ASCII '0'.

ASCII digit '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 trailing null.

ASCII digit '2' is 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.

ASCII digits '3' and '4' represent character special files and block special files respectively. In this case the *devmajor* and *devminor* fields shall contain implementation-defined information defining the device. Implementations may map the device specifications to their own local specification, or may ignore the entry.

ASCII digit '5' specifies a directory or sub-directory. 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.

ASCII digit '6' specifies a FIFO special file. Note that the archiving of a FIFO file archives the existence of this file and not its contents.

ASCII digit '7' is 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').

ASCII letters 'A' through 'Z' are reserved for custom implementations. All other values are reserved for specification in future revisions of the standard.

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 ASCII 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 References. <grp.h> §9.2.1, **<pwd.h>** §9.2.2, **<sys/stat.h>** §5.6.1, *stat*() §5.6.2, **<unistd.h>** §2.10.

10.1.2 Extended cpio **Format.** The byte-oriented cpio archive format is a series of entries, each comprised of a header that describes the file, the name of the file, and then the contents of the file.

An archive may be recorded as a series of fixed size blocks of bytes. This blocking shall be used only to make physical I/O more efficient. The last group of blocks is always at the full size.

For the byte-oriented cpio archive format, the individual entry information must be in the order indicated and described by Table 10-2.

Table 10-2. Byte-Oriented `cpio` Archive Entry

Header

Field Name	Length (in bytes)	Interpreted as
c_magic	6	octal number
c_dev	6	octal number
c_ino	6	octal number
c_mode	6	octal number
c_uid	6	octal number
c_gid	6	octal number
c_nlink	6	octal number
c_rdev	6	octal number
c_mtime	11	octal number
c_namesize	6	octal number
c_filesize	11	octal number

File Name

Field Name	Length	Interpreted as
c_name	*c_namesize*	pathname string

File Data

Field Name	Length	Interpreted as
c_filedata	*c_filesize*	data

10.1.2.1 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 ASCII characters interpreted as octal numbers. The octal numbers are extended to the necessary length by appending ASCII 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 which 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 implementation-defined manner.

(3) *c_mode* shall contain the file type and access permissions as defined in the tables below.

(4) *c_uid* shall contain the user ID of the owner.

(5) *c_gid* shall contain the group ID of the group.

(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 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 ASCII. For maximum portability between implementations, names should be selected from characters represented by the **portable filename character set** §2.3 as 8-bit characters with zero parity. If an extended character set beyond the portable character set is used, and the format-reading and format-creating utilities on the two distinct systems use the same extended character set, the file name shall be preserved. 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 standard; see *open*() §5.3.1, *stat*() §5.6.2, *chdir*() §5.2.1, *fcntl*() §6.5.2, and *opendir*() §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 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 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 Special Entries. FIFO special files, directories, and the trailer are recorded with *c_filesize* equal to zero. For other special files, *c_filesize* is implementation-defined. 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 an IEEE Std 1003.1-1988 conforming system; 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 standard 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

Table 10-3. Values for `cpio` *c_mode* Field

File Permissions

Name	Value	Indicates
C_IRUSR	000400	Read by owner
C_IWUSR	000200	Write by owner
C_IXUSR	000100	Execute by owner
C_IRGRP	000040	Read by group
C_IWGRP	000020	Write by group
C_IXGRP	000010	Execute by group
C_IROTH	000004	Read by others
C_IWOTH	000002	Write by others
C_IXOTH	000001	Execute by others
C_ISUID	004000	Set uid
C_ISGID	002000	Set gid
C_ISVTX	001000	Reserved

File Type

Name	Value	Indicates
C_ISDIR	040000	Directory
C_ISFIFO	010000	FIFO
C_ISREG	0100000	Regular file
C_ISBLK	060000	Block special file
C_ISCHR	020000	Character special file
C_ISCTG	0110000	Reserved
C_ISLNK	0120000	Reserved
C_ISSOCK	0140000	Reserved

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 standard. If the implementation does not support those flags, they may be ignored.

10.1.2.6 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.

Appendices

(These appendices are not a part of IEEE Std 1003.1-1988, IEEE Standard Portable Operating System Interface for Computer Environments, but are included for information only.)

A. Related Standards

This appendix describes other standards efforts, related to IEEE Std 1003.1-1988, that are available or under development.

A.1 Related Standards—Open System Environment. This IEEE Std 1003.1-1988 is intended to complement others that together would provide a comprehensive Open System Environment. The standards in these areas fall into three areas: ones directly related to the IEEE Std 1003.1-1988, ones already available and of use to those interested in Open Systems Environments, and finally, those in development.

IEEE and ANSI/IEEE standards can be ordered from:

Publication Sales
IEEE Service Center
P.O. Box 1331
445 Hoes Lane
Piscataway, NJ 08854-1331
(201) 981-0060

The document X3/SD-4 provides a list of all active X3 and related ISO projects, including approved standards. X3/SD-4 is available from:

CBEMA
X3 Secretariat
311 First Street, NW Suite 500
Washington, DC 20001-2178
(202) 737-8888

ANSI and ISO standards can be ordered from:

Sales Department
American National Standards Institute
1430 Broadway
New York, NY 10018
(212) 642-4900

A.2 Standards Closely Related to the 1003.1 Document.

A.2.1 System Interface. The 1003.1 Working Group will be continuing work in the area of system interface functions, preparing a supplement to IEEE Std 1003.1-1988. Tasks this group will be addressing include:

(1) Language-independent service specifications.

(2) An enhanced and unified data interchange format.

(3) Various additional functions suggested as a result of this standard.

Contact the IEEE Standards Office to participate in this effort. If you are interested in participating in this effort, or receiving working drafts and group mailings, contact the IEEE Standards Office; the address is listed in the **Foreword**.

A.2.2 C Language Standard. This document refers to the C Language Standard effort presently under development by Technical Committee X3J11 of the Accredited Standards Committee X3—Information Processing Systems. The X3J11 and 1003.1 groups have been cooperating to ensure that the standards are complementary and not overlapping. At the time of publication, the most recent X3J11 material was the version for public comment of the *ANSI/X3.159-198x Programming Language C Standard*, available from:

Global Engineering Documents, Inc.
2805 McGaw Street
Irvine, CA 92714
(800) 854-7179
(714) 261-1455
Telex: 692 373

Once the X3J11 document is approved, it will be available from the ANSI address given previously.

A.2.3 Shell and Utilities. This area is currently in development by IEEE Computer Society Working Group 1003.2. The proposed 1003.2 standard defines a source code level interface to shell services and common utility programs for application programs conforming to IEEE Std 1003.1-1988.* The proposed standard is being designed to be used by both application programmers and system implementors.

The following goals have been established for the Working Group:

Specify a standard interface that may be accessed in common by both applications programs and user terminal-controlling programs to provide services of a more complex nature than the primitives provided by IEEE Std 1003.1-1988. This interface shall be implementable on conforming IEEE Std 1003.1-1988

---

* An IEEE Std 1003.1-1988 conforming *implementation* is not necessarily required to support these application programs. Implementations could be produced that are conformant only to those 1003.1 features required by the proposed 1003.2 standard, and that cannot claim full conformance to all of IEEE Std 1003.1-1988.

systems. It shall include the following components:

(1) Application program primitives to specify instructions to an implementation-defined "shell" facility.

(2) A standard command language for a shell that includes program execution, I/O redirection and pipelining, argument handling, variable substitution and expansion, and a series of control constructs similar to other high-level structured programming languages.

(3) A recommended command syntax for command naming and argument specification.

(4) Primitives to assist applications programs and the shell language in parsing and interpreting command arguments.

(5) Recommended environment variables for use by shell scripts and application programs.

(6) A minimum directory hierarchy required for the shell and applications.

(7) A group of utilities that may be called from application programs for complex data manipulation and other tasks common to many applications.

(8) An optional group of utilities to be used for the software development of applications. The Working Group is examining both C language and FORTRAN software development requirements.

(9) Utilities and standards for the installation of applications.

The following areas are outside the scope of this standard:

(1) Operating system administrative commands (privileged processes, system processes, daemons, etc.).

(2) Commands required for the installation, configuration, or maintenance of operating systems or file systems.*

(3) Networking commands.

(4) Terminal control or user-interface programs (visual shells, window managers, command history mechanisms, etc.).

(5) Graphics programs or interfaces.

(6) Text formatting programs or languages.

(7) Database programs or interfaces (for example, SQL, etc.).

The Working Group is considering an expansion of its scope to include a "User Portability Extension," comprised of utilities which are used primarily by online users, such as full-screen editors. This subject area may be issued as a supplement to the first edition of the standard.

At the time of this printing, no approved document existed. Working drafts were being circulated, with a target schedule of mid-1989 for balloting.

Contact the IEEE Standards Office to participate in this effort.

---

* This is contrasted against paragraph **9**, above, by its orientation to installing the operating system itself, versus application programs. The exclusion of operating system installation facilities should not be interpreted to mean that the non-privileged application installation procedures *cannot* be used for installing operating system components.

A.2.4 Verification Testing. This area is currently in development by IEEE Computer Society Working Group 1003.3. A working draft is nearing completion for test methods related to IEEE Std 1003.1-1988; further standards that relate to the other working groups in 1003 are under study.

Contact the IEEE Standards Office to participate in this effort.

A.2.5 Realtime Extensions. This area is currently in development by IEEE Computer Society Working Group 1003.4, with a charter to develop and ballot extensions to IEEE Std 1003.1-1988 to address service interfaces needed for portable realtime applications.

The scope of the project includes the following realtime topics:

(1) High Resolution Timers
(2) Priority Scheduling
(3) Semaphores
(4) Contiguous Files
(5) Inter-Process Message Passing
(6) Event Notification
(7) Memory Locking
(8) Asynchronous I/O
(9) Synchronous I/O

The Working Group is an outgrowth of the /usr/group Technical Committee Realtime Subcommittee, with whom it holds joint meetings. At the time of this printing, no published document existed. Working drafts were being circulated.

Contact the IEEE Standards Office to participate in this effort.

A.2.6 Ada Language Bindings. This area is currently in development by IEEE Computer Society Working Group 1003.5, with a charter to develop and ballot extensions to IEEE Std 1003.1-1988 to provide an Ada language binding specification to the appropriate operating system interfaces.

Contact the IEEE Standards Office to participate in this effort.

A.2.7 Trusted System Extensions. This area is currently in development by IEEE Computer Society Working Group 1003.6, with a charter to develop and ballot extensions to IEEE Std 1003.1-1988 to address service interfaces needed for trusted, or high security systems. The Working Group is an outgrowth of the /usr/group Technical Committee Security Subcommittee, with whom it holds joint meetings.

Contact the IEEE Standards Office to participate in this effort. See also **Trusted Systems** §A.4.2.

A.2.8 Open System Guidelines. This area is currently in development by IEEE Computer Society Working Group 1003.0.

Contact the IEEE Standards Office to participate in this effort.

A.2.9 System Administration Extensions. A new working group is being formed in IEEE to cover POSIX system administration issues.

Contact the IEEE Standards Office to participate in this effort.

A.2.10 Networking Standards. A new working group is being formed in IEEE to cover POSIX networking-related issues. Contact the IEEE Standards Office to participate in this effort.

The ISO/OSI (Open System Interconnect) networking specifications are available from CBEMA or ANSI (and 802.*n* from the IEEE Standards Office):

OSI Model		ISO 7498 (ANSI)
Layer 1	CSMA/CD	IEEE 802.3 (IEEE)
	Token Bus	IEEE 802.4 (IEEE)
	Token Ring	IEEE 802.5 (IEEE)
Layer 2	Link Layer Control	IEEE 802.2 (IEEE) CCITT DR X.212 (CBEMA)
Layer 3	Network Layer	ISO 8348, 8473, 7777 (CBEMA)
Layer 4	Transport Layer	ISO 8072, 8073 (CBEMA)
Layer 5	Session Layer	ISO 8326, 8327 (CBEMA)
Layer 6	Presentation Layer	ISO DP 8822, DP 8823 (CBEMA)
Layer 7	Applications Layer	
	CASE (Common Services)	ISO DP 8649, DP 8650 (CBEMA)
	FTAM (File Transfer)	ISO DP 8571 (CBEMA)
	Mail/Message	CCITT X.400 series (CBEMA)
	Job Transfer	ISO DP 8831, DP 8832 (CBEMA)
Wide Area Net	Layers 1-3	CCITT X.25 (CBEMA)

A.2.11 Language Standards. The following language standards are available from ANSI:

Ada	Mil Std 1815-A-1983
Basic	X3.113-1987
COBOL	X3.23-1985
FORTRAN	X3.9-1978
Mumps	MDC X11.1-1984
Pascal	X3.97-1983

A.2.12 Graphics Standards. The following graphics-related standards are available from CBEMA or ANSI:

GKS	X3.124-1985 *Graphical Kernel System*; C language bindings are in progress (0533-D). (ANSI)
PHIGS	X3.144-198*x* *Programmers' Hierarchical Interactive Graphics System*; C language bindings are in progress (0534-D). (CBEMA)
CGM	X3.122-1986 *Computer Graphics Metafile*, formerly known as VDM, Virtual Device Metafile. (CBEMA)
X3H3.6	This working group is addressing windowing standards and display management for graphical devices. (CBEMA)

A.2.13 Database Standards. The following database standards are available from ANSI:

NDL	X3.133-1986 *Database Language NDL*. (Network Databases.)
SQL	X3.135-1986 *Database Language SQL*. (Relational Databases.)

A.3 Industry Open Systems Publications. The following publications describe recommendations formed by industry groups (as opposed to a single company) about related standards efforts.

The *X/OPEN Portability Guide* III (multiple volumes):

Prentice-Hall
200 Old Tappan Road
Tappan, NJ 07675

Reports of the /usr/group Technical Committees:

/usr/group
4655 Old Ironsides Drive #200
Santa Clara, CA 95054

Applications Environment Specifications published by the Open Software Foundation:

Open Software Foundation
20 Ballard Way
Lawrence, MA 01843

A.4 US Government Standards.

A.4.1 Federal Information Processing Standards (FIPS). Standards designated by the US Government as Federal Information Processing Standards frequently refer back to standards listed above.

For copies of POSIX-related FIPS documents and the NBS PCTS (POSIX FIPS Conformance Test Suite), contact:

National Technical Information Service
US Department of Commerce
5285 Port Royal Road
Springfield, VA 22161
(703) 487-4650

The Interim FIPS on POSIX announced in April of 1988 is based on Draft 12 of this document, which means that it differs in a few significant ways from this final standard. NBS has announced its intention that these differences will be eliminated in the next version of the FIPS, expected in late 1988.

A.4.2 Trusted Systems. A standard for secure, or trusted, systems, the *Department of Defense Trusted Computer System Evaluation Criteria*, Department of Defense Standard DoD 5200.28-STD, December 1985, is available from:

Office of Standards and Products
National Computer Security Center
Fort Meade, MD 20755-6000
Attn: Chief, Computer Security Standards

B. Rationale and Notes

This appendix summarizes the deliberations of the IEEE 1003.1 Working Group, the committee charged by IEEE with devising an interface standard for a portable operating system interface for computer environments, IEEE Std 1003.1-1988.

This appendix is derived in part from copyrighted draft documents developed under the sponsorship of /usr/group*, as part of an ongoing program of that association to support the IEEE 1003 standards program efforts.

The appendix is being published along with the standard to assist in the process of review. It contains historical information concerning the contents of the standard and why features were included or discarded by the Working Group. It also contains notes of interest to application programmers on recommended programming practices, emphasizing the consequences of some aspects of the standard that may not be immediately apparent.

B.1 Introduction. The IEEE Std 1003.1-1988 is based on the UNIX operating system developed by AT&T Bell Laboratories, and derives from efforts of the Standards Committee of /usr/group, an association of individuals, corporations, and institutions with an interest in the UNIX system that has long worked toward the development of independent industry-driven standards. The IEEE 1003 Working Group represents a cross-section of the UNIX system community: it consists of over 450 members representing hardware manufacturers, vendors of operating systems and other software development tools, software designers, consultants, academics, authors, applications programmers, and others. In the course of its deliberations, it has reviewed related American and international standards, both published and in progress.

Conceptually, this standard describes a set of fundamental services the 1003 Working Group feels are needed for the efficient construction of application programs. Access to these services has been provided by defining an interface, using the C programming language, which 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 dubbed POSIX, an acronym for Portable Operating System Interface. The name POSIX, suggested by Richard Stallman, was adopted during the printing of the Trial Use Standard.

Although originally coined by the IEEE to refer to IEEE Std 1003.1-1988, the term POSIX more correctly refers to a *family* of related standards or working groups, 1003.*n*. These other activities are described in Appendix A. There are some cases where this Rationale (and the standard itself) uses the term POSIX as a synonym for IEEE Std 1003.1-1988. This incorrect usage is maintained for purposes of readability only.

As explained in the Foreword, the term POSIX is expected to be pronounced *pahz-icks*, as in *positive*, not *poh-six*, or other variations. The 1003 Working Group has published the pronunciation of its term in an attempt to promulgate a standardized way of referring to a standard operating system interface.

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

(1) persons buying hardware and software systems;

(2) persons managing companies that are deciding on future corporate computing directions;

(3) persons implementing operating systems, and especially;

(4) persons developing applications where portability is an objective.

B.1.1 Scope. This Rationale focuses primarily on additions, clarifications, and changes made to the UNIX system as described in the **Base Documents** §B.1.3 from which the standard was derived. It is not a rationale for the UNIX system as a whole, since the Working Group was charged with codifying existing practice, not designing 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 which is "outside the scope" or otherwised not addressed by this standard is implicitly "implementation-defined" if it is included in an implementation.

The Rationale discusses some UNIX system features that were *not* adopted into the standard. 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 the standard. It is hoped that this Rationale can provide appropriate answers.

There are choices allowed by the standard for some details of the interface specification; some of these are specifiable optional subsets of the standard. See **Symbolic Constants** §B.2.10. See also **Specific Derivations** §B.1.3.3.

After deliberation, the 1003.1 Working Group decided that although the services this standard 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 prompted the Working Group to coin the term *IEEE Std 1003.1-1988 with the C Language Binding*. The word *Binding* refers to the binding of a conceptual set of services and a standardized C interface which establishes rules and syntax for accessing them. Forthcoming extensions to this standard are expected to include bindings for other programming languages.

The current version of the C Standard will be the basis for functional definitions of core services that are independent of programming languages. The standard as it stands now can be thought of as a C Language Binding. Chapters 1 through 7, and 9, correspond roughly to the C language implementation of what will be defined in the programming language-independent core services section of the standard; Chapter 8 corresponds to be the C language-specific section.

Readers are warned that the criteria used to choose the programming language-independent core services may be different from expected. The core services represent services that are common to those programming languages likely to form language bindings to this standard—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 this section. 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 the standard 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.

Implementors and programmers can rest assured, however, that any programming language traditionally supporting memory management will include those routines in the language-dependent sections 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 sections 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 **Language-Dependent Services for the C Programming Language** §B.2.2.3.

The standard 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 books on the UNIX System in **Historical Implementations** §B.11.2.

B.1.2 Purpose. Several principles guided the Working Group's decisions.

B.1.2.1 Application Oriented. The basic goal of the Working Group was to promote portability of application programs across UNIX system environments by developing a clear, consistent, and unambiguous standard for the interface specification of a portable operating system based on the UNIX system documentation. This standard codifies the common, existing definition of the UNIX system. There was no attempt to define a new system interface.

B.1.2.2 Interface, Not Implementation. The standard defines an interface, not an implementation. No distinction is made between library functions and system calls: both are referred to as functions. No details of the implementation of any function are given (although historical practice is sometimes indicated in the Rationale). Symbolic names are given for constants (such as signals and error numbers) rather than numbers.

B.1.2.3 Source, Not Object, Portability. The standard has been written so that a program written and translated for execution on one conforming implementation may also be translated for execution on another conforming implementation. The standard does not guarantee that executable (object or binary) code will execute under a different conforming implementation than that for which it was translated, even if the underlying hardware is identical. The Working Group has, however, attempted to put few impediments in the way of binary compatibility, and some remarks are found in this Rationale. See **Requirements** §B.2.2.1.1 and **Configurable System Variables** §B.4.8.

B.1.2.4 The C Language and X3J11. The standard is written in terms of the standard C language as specified in the C Standard (*ANSI/X3.159-198x Programming Language C Standard*) produced by the X3J11 Working Group. See **Conformance** §2.2. Guidelines used in negotiations between the two Working Groups are discussed below in **POSIX and the C Standard** §B.1.4.

B.1.2.5 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 the standard, and functions usable only by the super-user have not been included. This Rationale notes several such instances. Still, an implementation of the standard interface may also implement features not in the standard: see **Requirements** §2.2.1.1. The standard is also not concerned with hardware constraints or system maintenance.

B.1.2.6 Minimal Interface, Minimally Defined. In keeping with the historical design principles of the UNIX system, the standard 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*() §5.3.2. In such cases, as throughout the standard, redundant definitions were avoided: *creat*() §5.3.2 is defined as a special case of *open*() §5.3.1. Redundant functions or implementations with less tradition were excluded. For example, *seekdir*() and *telldir*() were not included in **Directory Operations** §5.1.2.

B.1.2.7 Broadly Implementable. The Working Group has 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 AT&T code (Version 7 or later).
(2) Compatible systems that are not derived from AT&T code.
(3) Emulations hosted on entirely different operating systems.
(4) Networked systems.
(5) Distributed systems.
(6) Systems running on a broad range of hardware.

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

B.1.2.8 Minimal Changes to Historical Implementations. There are no known **Historical Implementations** §B.1.3.2 that will not have to change in some area to conform to the standard, and in a few areas the standard does not exactly match any existing system interface (for example, see **O_NONBLOCK** §B.6). Nonetheless, there is a set of functions, types, definitions, and concepts that form an interface that is common to most historical implementations. The standard 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** §5.1, or;

(2) by specifying an interface that is readily implementable in terms of, and backwards compatible with, existing implementations, such as **Extended tar Format** §10.1.1, or

(3) by specifying an interface that, when added to an historical implementation, will not conflict with it, like **O_NONBLOCK** §B.6.

Required changes to historical implementations have been kept as few as possible, but they do exist, and this Rationale points out some of them.

The standard is specifically not a codification of a particular vendor's product. It is like the UNIX system, but it is not identical to it. The word UNIX is not used in the standard proper both for that reason, and because it is a trademark of a particular vendor.

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 "deprecated" and the standard functions used for new applications. Some extensions will represent functions beyond the scope of POSIX. These need to be used with careful management to be able to adapt to future POSIX Extensions, and/or port to implementations that provide these services in a different manner.

B.1.2.9 Minimal Changes to Existing Application Code. The Working Group wished to make less work for application developers, not more. However, because every known historical implementation will have to change at least slightly to conform, some applications will have to change. This Rationale points out the major places where the standard implies such changes.

B.1.2.10 IEEE Consensus Process. The IEEE consensus process was used in deliberations. There are several levels of participation:

(1) **Correspondents.** Those interested in following the development of the standard could subscribe to a mailing list to which copies of drafts, working documents, and related material were sent. Also, anyone (including individuals, companies, government agencies, or other organizations) could send comments (or RFCs, Proposals, or Notes) to the Working Group.

(2) **Working Group.** This was the group responsible for producing the standard document. It met four times a year and produced many drafts. It also produced the Trial Use and Full Use Standards, and was responsible for resolving balloting objections to them. The Working Group was composed of individuals, even though many of them worked for companies with interests

in the field.

(3) **Balloting Group.** This group voted on the proposed standards in the manner detailed in the next subsection. The Balloting Group, like the Working Group, was composed of individuals. Most of the people on the Working Group also were in the Balloting Group, although the latter included many others, as well.

(4) **Institutional Representatives.** Exceptions to the individual composition of the Balloting Group were the Institutional Representatives, who represented related standards bodies or professional organizations (in this case, USENIX, /usr/group, and X/Open).

Decisions of the Working Group were not made by vote, not even of a large majority. Decisions were made by consensus, which required that every individual believe that:

(1) their point of view had been heard.

(2) their point of view had been understood.

(3) other individuals' points of view were adequately understood.

(4) there was general consensus.

A common way of moving discussion along was to ask if anyone would ballot "no" on a particular issue.

B.1.2.11 IEEE Balloting Process. The IEEE balloting process was used to attain the ANSI requirement for a consensus acceptance of a document as a standard.

Balloting in IEEE is done by individuals who are members of IEEE or affiliated with the IEEE Computer Society. They are given thirty days in which to return the ballots, and 75% of those in the balloting group must return ballots.

Ballots from non-IEEE members (or non-Computer Society affiliates) are also included in the process, with comments and objections treated the same as those from members. However, non-IEEE members are not included in the percentages of returns required or the affirmative percentage required for approval. Possible ballot responses (excluding abstentions) are:

(1) **yes without comments.**

(2) **yes with comments.** The comments indicate areas that should be evaluated, but are not significant enough to warrant a negative ballot.

(3) **no with objections.** A negative ballot must include specific objections and recommendations on how to resolve the objections. These objections indicate areas that must be fixed to resolve the negative ballot.

At least 75% of those balloting (not abstaining) must provide an affirmative response. Each objection, and many of the comments, are translated into proposed changes; and any outstanding objections, along with the rationale for not making the changes to accommodate these objections, are fed back to the balloting group.

Members of the balloting group are given ten (or more) days to change their ballots, with similar options as above; however, objections are limited to the proposed changes and/or failure to resolve key objections. It is possible for the number of negative responses to increase if a proposed change is objectionable, or if a significant objection has not been addressed.

In general, the balloting process moves fairly quickly towards a high degree of consensus. The final results are submitted to the IEEE Standards Board for approval, and include the balloting percentages as well as documentation of any unresolved negative objections.

The Institutional Representatives were exceptions in several ways.

(1) They are not required to be IEEE members.

(2) They ballot for their Institutions, not as individuals.

(3) Ballots of Institutional Representatives are reported separately to the IEEE Standards Board.

As with other ballots, any unresolved negative objections are reported with the rationale for not incorporating the associated changes. However, the separate reporting of the Institutional ballots tends to make any objections more visible, particularly in that Institution's areas of expertise; consequently, any unresolved objection could be enough to cause the document to be sent back to the balloting process for further resolution.

The Trial Use period was from April 1986 to November 1987, when the balloting of the revised document (Draft 12) began, and provided an additional level of industry consensus. The high visibility of the document, as well as its widespread distribution, provided additional feedback and information for the formulation of the current standard. See also **Specific Derivations** §B.1.3.3.

B.1.2.12 WeirdNIX ... *or destructive QA of a standard.*

As part of the evaluation of the Trial Use Standard, the IEEE 1003 Working Group offered prizes for the best new and technically legal interpretation of the POSIX standard which nevertheless violates the intuitive intent of the POSIX standard.

The intent was to find how the standard might be misinterpreted, and then to correct the errors that the misinterpretations point out. Like destructive testing of hardware, you stress it until it breaks and then fix what broke so you can then stress it further.

The criteria for judging the misinterpretations were:

(1) It had to be an interpretation of the P1003.1 POSIX Trial Use standard (as published by IEEE) which conforms completely to the standard. For the purposes of the contest, Appendices C.5, E.1 and J were included as part of the standard, but no other Appendices.

(2) It had to be represented as a detailed description in either pseudo-code and/or text as how an implementation could behave so as to conform with the standard and still do "the wrong thing." Annotation as to why the interpretation is considered legal by the submitter was of significant value in judging.

(3) Interpretations had to be of topics discussed in the standard. Areas that are not covered by the standard are not eligible. Interpretations which use some features of the standard and then take advantage of something upon which the standard is silent (and thus should not be) were of significant value.

The winners were judged by a subset of the IEEE 1003 Working Group.

Prizes were awarded to the *Best* and *Most Demented* interpretations. *Best* is an interpretation that is legal and which is "likely," in that one could reasonably

make the mistake and implement a system which did that. *Most Demented* is a legal interpretation that would not actually be implemented because it violates common sense.

The prizes were:

(1) HP 16C calculators, donated by Hewlett-Packard,

(2) Having the winners' names in a place of honor in the IEEE Std 1003.1-1988 final use standard,

(3) A copy of the final use standard.

The winners were announced at the USENIX conference in January 1987 and are recognized in the **Foreword** of this standard.

WeirdNIX Post Mortem: The results of the contest indicated that the idea was probably sound, but the lack of available copies of the standard and the short contest period (dictated by meeting schedules) made it less successful than it could have been. Other standards, particularly in the computer area, might wish to consider trying this again with better planning.

B.1.3 Base Documents. The Working Group consulted a number of documents as representing features appropriate for consideration for inclusion in the standard. Full bibliographic information may be found in **Bibliographic Notes** §B.11.

B.1.3.1 Related Standards and Documents.

(1) *1984 /usr/group Standard*

(2) *ANSI/X3.159-198x Programming Language C Standard*

(3) *X/OPEN Portability Guide*

The most direct ancestor is the *1984 /usr/group Standard*, which is considered to be Draft 1 of the present standard. It, in turn, was largely derived from the programming interface of System III. The *1984 /usr/group Standard* is also the principal ancestor of the Library section of the C Standard.

The X3J11 and 1003.1 Working Groups cooperated closely. Details of the relations of the two standards they produced are listed in this Rationale in **POSIX and the C Standard** §B.1.4 because the C Standard is the standard most closely related to POSIX. POSIX is written in terms of the C Standard, although it is possible to have POSIX without Standard C: see **Conformance** §B.2.2.

The *X/OPEN Portability Guide* proved useful because X/Open had in many cases already addressed the same issues as P1003.1, though often in a slightly different context.

The Working Group was aware of the Japanese SIGMA Project, which includes as a goal a common operating system interface specification, and there was a representative of SIGMA at many 1003.1 Working Group meetings.

B.1.3.2 Historical Implementations. These include (with colloquial names in parentheses):

(1) UNIX Time-Sharing System: UNIX Programmer's Manual, Seventh Edition (Version 7)

(2) UNIX System III Programmer's Manual

(3) AT&T System V Interface Definition (SVID), Issue 2, Volumes 1-3

(4) 4.3 Berkeley Software Distribution, Virtual VAX-11 Version (4.3BSD)

Manuals

The UNIX system has changed more since the *1984 /usr/group Standard* was written than has the C language, and there are more variants of the former. Because of this, the present standard was radically reorganized and reformatted after the first draft and had many changes in content. Thus there is no single Base Document to provide context for all discussions in this Rationale, which instead discusses aspects of Version 7, System III, System V, and 4.3BSD that were included in this standard or that were considered in choosing what was included.

Occasional mentions are made of the Eighth and Ninth Editions, which are successors of Version 7, the Bell Laboratories research system. The context is usually related to the *streams* inter-process communication mechanism, which is not in this standard but which has influenced discussions about inter-process communication mechanisms.

Although 4.2BSD was the current Berkeley Software Distribution when most of the work on the standard was done, this Rationale refers to 4.3BSD instead (in most places) because the differences between the two versions are almost entirely in performance, the few programming interface differences are mostly outside the scope of this standard, and the 4.3BSD manuals actually describe 4.2BSD better than the 4.2BSD manuals do.

The System V manuals are never referenced because the *SVID* is more definitive.

Much of the standard is closer to the *SVID* than to any other document.

Parts of documentation of many other related systems were considered in deliberations on various aspects of the standard. As those were too numerous to list all of them, none of them will be mentioned by name.

B.1.3.3 Specific Derivations. Some areas of the standard are clearly derived from facilities of specific systems. Most of the major areas are listed here, together with references to the sections of the standard where they occur. For most of them, there is also more detail in the corresponding sections of the Rationale.

FIFOs
: The **FIFO special file** §2.3 facility exists in System III, the *1984 /usr/group Standard*, and System V, but not in Version 7, 4.2BSD, or 4.3BSD.

reliable signals
: **Signals** §3.3 includes reliable signals related to the 4.3BSD model. These were introduced to address concerns with signal reliability that were raised in balloting the Trial Use Standard (thus the name "reliable signals").

job control
: The **job control** §B.3.3 facility is derived from 4.3BSD and was introduced between the Trial Use and Full Use Standards.

saved set-user-ID (saved set-group-ID)
: This optional capability, mostly in *exec* §3.1.2 and **Set User and Group IDs** §4.2.2, is derived from System V, and was introduced in the Trial Use Standard.

supplementary groups
: A single group per process as in System V is the default, but

User Identification §4.2 (particularly *getgroups*() §4.2.3) allows multiple groups per process as in 4.3BSD as an option. This was introduced shortly before the Trial Use Standard.

uname() The *uname*() §4.4.1 function is derived from the *1984 /usr/group Standard*, which took it from System III, and it is still in System V. It does not exist in Version 7 or 4.3BSD.

opendir(), *readdir*(), *rewinddir*(), *closedir*()
The section on **Directory Operations** §5.1 is derived from 4.2BSD and was introduced in an early draft of the standard. It was later adopted in System V Release 3.

mkdir(), *rmdir*(), *rename*()
The three functions *mkdir*() §5.4.1, *rmdir*() §5.5.2, and *rename*() §5.5.3 are derived from 4.2BSD. Except for *rename*(), these functions now also appear in System V Release 3.

termios **Device- and Class-Specific Functions** §7, while closer to System V than to 4.3BSD, does not correspond to any existing system because none was found adequate when considerations such as international character sets, fast interfaces, and networks were taken into account. The final interface specification was introduced shortly before the Full Use Standard.

archive format
The **Extended tar Format** §10.1.1 is derived from the `tar` programs used in Version 7 and 4.3BSD, and provided with System V. The precise format in the Full Use Standard has evolved incrementally from that in earlier drafts of POSIX. The **Extended cpio Format** §10.1.2 is derived from that of System V.

B.1.3.4 Working Documents. The model for the present Rationale was the Rationale prepared by the X3J11 Working Group to accompany the C Standard: *X3J11/86-152, October 1, 1986* "Rationale for Draft Proposed American National Standard for Information Systems—Programming Language C." Its influence may be seen most clearly in **POSIX and the C Standard** §B.1.4, but it also is present in more subtle ways throughout.

References to programs, functions, or facilities of systems described by the Base Documents (such as the System V `cpio` utility program) have been freely included in this Rationale where relevant, even though they would be inappropriate in the standard itself. References to programs, functions, or facilities not described by the base documents or to companies not directly associated with them have been excluded where possible. Exceptions have been made where facilities were derived from systems not described by the base documents, and where the word "may" is used to describe an option that permits behavior of such a system.

B.1.4 POSIX and the C Standard. Some C language functions and definitions were handled by POSIX, but most by X3J11's ANSI C Standard. The most general guideline was that POSIX retained responsibility for operating-system specific functions, while X3J11 defined C library functions. See also **C**

Language Definitions §B.2.8 and **Language-Specific Services for the C Programming Language** §B.8.

There are several areas in which the two standards differ philosophically:

(1) **Function parameter type lists.** These appear in the C Standard and specify the types of the arguments and return values of functions in external references to them. POSIX does not include them, except in a few places to indicate variable number of arguments, e.g., **File Control** §B.6.5.2. Function parameter type lists were not used because the Working Group was aware that some vendors would wish to implement POSIX in terms of a binding to an historical variant of the C language instead of to the C Standard, since compilers for the latter would initially not be widespread. Since the C Standard does not require the use of function parameter type lists, the function definitions used in POSIX are nonetheless specified in terms of Standard C. See also **Signals** §B.3.3.

(2) **Single vs. multiple processes.** The C Standard 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. But the POSIX interface is that of a multi-process timesharing system. Thus POSIX has to take multiple processes into account in places where the C Standard does not mention processes at all, such as *kill*() §3.3.2. See also **Requirements** §B.2.2.1.1.

(3) **Single vs. multiple operating system environments.** The C Standard 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 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** §B.1.2.7 in terms of various underlying operating systems. See also **Requirements** §B.2.2.1.1.

(4) **Translation vs. execution environment.** POSIX is primarily concerned with the Standard C *execution environment*, leaving the *translation environment* to the C Standard. See also **Requirements** §B.2.2.1.1.

(5) **Hosted vs. freestanding implementations.** All POSIX implementations are hosted in the sense of the C Standard. See also the remarks on conformance in the **Foreword**.

(6) **Text vs. binary file modes.** X3J11 defines *text* and *binary* modes for a file. But the POSIX interface and historical implementations related to it make no such distinction, and all functions defined by POSIX treat files as if these modes are identical. (It is important not to say that POSIX files are either *text* or *binary*.) X3J11 wrote their definitions 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.

And there is a basic difference in approach between the X3J11 Rationale and the POSIX Rationale. The X3J11 Rationale addresses almost all changes as differences from the Base Documents of the C Standard, usually either Kernighan and Ritchie (see **Related Standards** §B.11.1) or the *1984 /usr/group*

Standard. The present 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 X3J11 marks QUIET CHANGES from the Base Documents in its Rationale. The POSIX Rationale 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.

B.1.4.1 Solely by POSIX. These return parameters from the operating system environment: *cuserid*() §4.2.4, *ctermid*() §4.7.1, *ttyname*() §4.7.2, and *isatty*() §4.7.2.

The functions *fileno*() §8.2.1 and *fdopen*() §8.2.2 map between C language stream pointers and POSIX file descriptors.

B.1.4.2 Solely by X3J11. There are many functions that are useful with the operating system interface and are required for conformance with the present standard, but that are properly part of the C Language. These are listed in **Referenced C Language Routines** §8.1, which also notes which functions are defined by both POSIX and X3J11. Certain terms defined by X3J11 are incorporated by POSIX in **C Language Definitions** §2.8.

Some routines were considered too specialized by the 1003.1 Working Group to be included in the standard. These include *bsearch*() and *qsort*().

B.1.4.3 By Neither POSIX Nor X3J11. Some functions were considered of marginal utility and problematical when international character sets were considered: *_toupper*(), *_tolower*(), *toascii*(), and *isascii*().

Though *malloc*() §8.1 and *free*() §8.1 are in the C Standard and are required by **Referenced C Language Routines** §8.1 of the present standard, neither *brk*() nor *sbrk*() occur in either standard (although they were in the *1984 /usr/group Standard*), because this standard is designed to provide the basic set of functions required to write a Conforming POSIX Application; the underlying implementation of *malloc*() or *free*() is not an appropriate concern for the standard.

B.1.4.4 Base by POSIX, Additions by X3J11. Since the C Standard does not depend on POSIX in any way, there are no items in this category.

B.1.4.5 Base by X3J11,Additions by POSIX. X3J11 has to define *errno* if only because examining that variable is the only way to tell when some mathematics routines fail. But POSIX uses it more extensively, and adds some semantics to it in **Error Numbers** §2.5, which also defines some values for it.

Many numerical limits used by X3J11 were incorporated by POSIX in **Numerical Limits** §2.9, and some new ones are added, all to be found in the header **<limits.h>**.

The POSIX definition of **signal** §2.3 further specifies the C definition, and the entire mechanism of **Signals** §3.3 is much more elaborate.

The function *time*() §4.5.1 is used by X3J11, but POSIX further specifies the time value.

The function *getenv*() §4.6.1 is referenced in **Environment Description** §2.7 and *exec* §3.1.2 and is also defined by X3J11.

The function *rename*() §5.5.3 is extended to further specify its behavior when the new filename already exists or either argument refers to a directory.

B.1.4.6 Related Functions by Both. The X3J11 definition of compliance and the POSIX definition of **Conformance** §2.2 are similar, although the latter notes certain potential hardware limitations.

POSIX defined a portable filename character set in **General Terms** §2.3, that is like the X3J11 identifier character set. However, POSIX did not allow upper- and lowercase characters to be considered equivalent. See **filename portability** §2.4.

The *exit*() function is defined only by X3J11, because it refers to closing streams, and that subject, as well as *fclose*() itself, is defined almost entirely by X3J11. But POSIX defined *_exit*() §3.2.2, which also adds semantics to *exit*(). This also allows POSIX to ignore the X3J11 *atexit*() function.

POSIX defined *kill*() §3.3.2, while X3J11 defined *raise*(), which is similar except that it does not have a process ID argument, since the language defined by X3J11 does not incorporate the idea of multiple processes.

The new functions *sigsetjmp*() §8.3.1 and *siglongjmp*() §8.3.1 were added to provide similar functions to X3J11 *setjmp*() and *longjmp*() that additionally save and restore signal state.

B.1.5 Organization.

B.1.5.1 Organization of the Standard. See the **Foreword**.

It was decided very early that the traditional organization by manual section, as used in the *1984 /usr/group Standard*, would be confusing in an IEEE standard. That organization assumed some background that was not relevant to the purpose of the standard. It also made an implementation-oriented distinction between system calls and library routines, which were in separate sections.

Two sections, **Scope** §1 and **Definitions and General Requirements** §2, have been added because they are traditional in IEEE standards. A **Foreword** was added for the same reason, even though it is not part of the standard proper.

Although appendices were used in the Trial Use Standard to contain proposals for examination by the Balloting Group and the general public, the Full Use Standard has no proposal appendices, because the text of the standard proper must be complete. The Appendices of the Full Use Standard discuss either related standards or the Full Use Standard itself. The Full Use Standard contains some new material that was not in the Trial Use Standard, mostly that which was added to meet balloting objections. The most obvious examples are the addition of reliable signal considerations to **Signals** §3.3 (including the addition of **Non-Local Jumps** §8.3.1) and the resolution of **Device- and Class-Specific Functions** §7. See also **Specific Derivations** §B.1.3.3.

Because there were too many notes interpolated in the text of the Trial Use Standard (which were nonetheless not part of the standard), and because there were still not enough to explain why the Working Group had made many difficult decisions, the Working Group decided to add a Rationale and Notes Appendix, modeled after the one the X3J11 Working Group was producing for the C Standard. Most of the notes formerly in the main body of the draft were moved to the Rationale appendix, although some were deleted and others were incorporated into the text of the standard proper.

B.1.5.2 Organization of this Appendix. Just as the standard proper excludes all examples, footnotes, references, and appendices, this Rationale is also not part of the standard. The POSIX interface is defined by the standard alone. If any part of this Rationale is not in accord with that definition, the IEEE Standards Office should be so informed. In the meantime, conflicts between this Rationale and the standard are always resolved in favor of the body of the standard.

All sections of this appendix beginning with **Definitions and General Requirements** §B.2 follow the exact structure of the standard, and aspects of a given section of the standard are considered in the corresponding section of the Rationale. Where a given discussion touches on several areas, attempts have been made to include cross-references within the text.

References to the standard are in the same format as references within the standard to parts of itself, for example: **General Terms** §2.3. References to this Rationale are given as references to Appendix B of the standard, that is, the section numbers always begin with "B." as in **General Terms** §B.2.3. Where a reference both to part of the standard and to a related note in the Rationale would be appropriate only the latter is given, because all parts of the Rationale implicitly refer to the corresponding parts of the standard.

B.1.5.3 Typographical Conventions. A summary of typographical conventions is shown in Table B-1.

Table B-1. Typographical Conventions

Reference	Example
Command Name	`cpio`
Data Types	*long*
Defined Terms	*file*
Environment Variables	**PATH**
Error Numbers	[EINTR]
Function Arguments	*arg1*
Functions	*open*()
Global Externals	*errno*
Headers	**<sys/stat.h>**
Limits	{OPEN_MAX}
Macros	S_ISDIR
Section References	**Process Termination** §3.2
Symbolic Constants	{_POSIX_VDISABLE} or O_NONBLOCK

Defined terms are shown in three styles, depending on context:

(1) The initial appearance of the term is in *italics*. In the section **General Terms** §2.3, every reference to a term is italicized, which highlights the interrelationships between the definitions.

(2) Subsequent appearances of the term are in the Roman font.

(3) Cross references to terms from outside section 2.3, such as **seconds since the Epoch** §2.3, are shown in **bold**, because that is the style for all

cross references to terms and headings.

Cross references to other section headings are not always exactly accurate. In this Rationale, in particular, keywords and other short phrases are sometimes used, rather than the full section heading text.

Macros are either uppercase, as shown, or lowercase, depending on their historical origins.

Symbolic constants are shown in two styles: those within braces are intended to call the reader's attention to values in **<limits.h>** and **<unistd.h>**; those without braces are normally defined by one or a few related functions.

In this Rationale, occasional use is made of an historical artifact from the UNIX system shell. In the shell, the asterisk (*) is used in file names or other command line tokens to represent zero or more additional characters. Thus, the construct **LC_*** represents all the environment variables beginning with the characters **LC_**.

Defined names that are normally in lowercase, particularly function names, are never used at the beginning of a sentence or anywhere else that normal English usage would require them to be capitalized.

The above typographical conventions apply to both the standard and to this Rationale. There are also some conventions peculiar to the Rationale, regarding standards for the operating system interface and for the C language. These are used frequently in **POSIX and the C Standard** §B.1.4 and are shown in Table B-2.

Table B-2. Short Name Usages

Topic	Operating System Interface	C Programming Language
Working Group	1003.1	X3J11
standard	IEEE Std 1003.1-1988	*ANSI/X3.159-198x Programming Language C Standard*
short name	POSIX	C Standard
Rationale	Appendix B	*Rationale for American National Standard for Information Systems—Programming Language C*
short name	this Rationale	X3J11 Rationale

The name POSIX is usually used for the IEEE Std 1003.1-1988 instead of the name 1003.1, because the latter is too easily confused with the name of the Working Group, 1003.1.

"Standard C" or the "C Standard" will eventually come to mean "ISO C," but currently refers to the *ANSI/X3.159-198x Programming Language C Standard* produced by the X3J11 Working Group.

B.1.5.4 Document Indexes. The document comes with two indexes that are produced by completely different methods:

(1) The *Identifier Index* is built from hand-inserted macros in the document text. It points to the definition of each of the functions, headers, global variables (such as *errno*), and *families* (groups of functions, such as the *exec* and *wait* families).

(2) The *Topical Index* is primarily the product of AT&T's Documenter's Workbench* software. A list of key symbols and phrases is stored in a file and a `yacc`-based program finds instances of these phrases in the document. Merged into the output of that process are all of the hand-inserted macros within the text, including definitions as well as the four categories of the Identifier Index. These merged macros cause the page number reference to be listed in bold face, such as:

index entry ... **57**, 22, 55-60, 105

Thus, the bold reference is probably the most interesting first stop for the index user. Other references may or may not be of interest, but are included for completeness. The reader is warned that it may be difficult to find some indexed phrases on the pages shown; the phrase scanner occasionally makes intuitive leaps that connect unrelated words in a sentence; it is also possible that a phrase close to the bottom of a page may be referenced on the following page number.

Of course, neither index is part of the standard and the presence or accuracy of terms and page references have no normative effect. Suggestions for correcting errors or improving the index are welcome; this same indexing software is used by all of the POSIX technical editors and many future standards may benefit. Address suggestions to the IEEE office listed in the **Foreword**, requesting that they be forwarded to the Technical Editor of the 1003 Working Group.

B.2 Definitions and General Requirements.

B.2.1 Terminology. The meanings specified in the standard for the words *shall*, *should*, and *may* are mandated by IEEE.

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; see **Organization of this Appendix** §B.1.5.2.

implementation-defined. This definition is analogous to that of the C Standard, and, together with *undefined* and *unspecified*, provides a range of specification of freedom allowed to the interface implementor.

---

* Documenter's Workbench is a trademark of AT&T, Inc.

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 the standard 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.

should. In this standard, 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.

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].

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 the POSIX standard 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. In an ideal world, there could be a fourth term, specifying "We don't say what this does; it's acceptable to define it in an implementation, but you don't need to document it," and undefined would then be used very rarely for the few things for which any definition is not useful.

B.2.2 Conformance. These conformance definitions are descended from those of *conforming implementation*, *conforming application*, and *conforming portable application*, of the Trial Use Standard, but were changed to clarify

(1) extensions, options, and limits;

(2) relations among the three terms, and;

(3) relations between POSIX and the C Standard.

B.2.2.1 Implementation Conformance. These definitions allow application developers to know what they can 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 the standard 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 the present standard.

B.2.2.1.1 Requirements. The word "support" is used rather than "provide" in order to allow an implementation that has no resident software development facilities, but which supports the execution of a *Strictly Conforming POSIX Application*, to be a *conforming implementation*. See also **Translation vs. Execution Environment** §B.1.4.

B.2.2.1.2 Documentation. The conforming documentation shall use the same numbering scheme as this standard for purposes of cross referencing. (This also eliminates the need for a definitive "laundry list.")

This proposal is consistent with and supplements the verification test suite developed by the 1003.3 Working Group. All options that an implementation chooses shall be reflected in **<limits.h>** and **<unistd.h>**.

Hardware Failures: 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 arbitrarily bad. 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 this standard. 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.2.2.1.3 Conforming Implementation Options. Within this standard there are some symbolic constants that, if defined, indicate that a certain option is enabled. Other symbolic constants exist in the standard for other reasons. This section was placed in the standard to help 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, the standard at one point permitted either of (typically) two options using "may." At the request of the 1003.3 Working Group, 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 Working Group 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.

The Working Group intends 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 enough, as this is not consistent with the intent of the standard: 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.2.2.2 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.

The standard (and this Rationale) occasionally use 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 Application Using Extensions, to implementation extensions. When the two looser expressions are used, it should be apparent from the context of the discussion which of the more formal names is appropriate.

B.2.2.2.1 Strictly Conforming POSIX Application. This definition is analogous to that of a Standard C *conforming program*.

The major difference between a *Strictly Conforming POSIX Application* and a Standard C *strictly conforming program* is that the latter is not allowed to use features of POSIX that are not in the C Standard.

B.2.2.2.2 Conforming POSIX Application. Examples of *<National Bodies>* include ANSI, BSI, and AFNOR.

B.2.2.2.3 Conforming POSIX Application Using Extensions. Due to possible requirements for configuration or implementation characteristics in excess of the specifications in **<limits.h>** §2.9 or related to the hardware (such as array size or file space), not every Conforming POSIX Application Using Extensions will run on every conforming implementation.

B.2.2.3 Language-Dependent Services for the C Programming Language. This standard is, for historical reasons, both a specification of an operating system interface and a C binding for that specification.

It is clear that these two need to be separated into separate entities, but the urgency of getting any 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 this standard (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 this standard. In this case, a mapping between the functionality in that language and the underlying functionality in this standard 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 this standard, 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 which 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 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 to do the same thing.

A program which requires the POSIX capabilities which are not bound to the standard language directly (as above) has no chance to be portable outside the POSIX environment. It doesn't matter whether the extension is syntactic or a new function; it still won't port without effort. Given this, it seems unreasonable not to consider language extensions when determining how best to map the functionality of POSIX 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.

It should be clear that no attempt to mimic *printf*() or *scanf*() (or the rest of the Standard C functions) should be made, but rather that the equivalent functions in the language should be used. (Formatted READ and WRITE in FORTRAN, `read/readln` and `write/writeln` in Pascal, for example.)

It should be noted that there is inherently a 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 have much likelihood of binding directly to statements in a general purpose language. However, an operating system standard should be providing the services required by a language, and there should be a good chance of this happening. This is an unusual situation, and the natural 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 too much to ask, but bindings such as those for standard I/O in Chapter 8 of this standard are quite reasonable.)

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, for the languages to request changes to the operating system to facilitate a better such mapping. (A future inclusion of a truncate function, specifically for the FORTRAN ENDFILE statement, but which is also generally useful, is a good example.)

Clearly, part of the job of creating a binding is choosing names for functions which are introduced, and these will need to be appropriate for that language. It may be reasonable 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 which are not POSIX, and if POSIX conformant systems typically accept names which 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 **Scope** §B.1.1 for additional information about C bindings.

B.2.2.3.1 Types of Conformance.

B.2.2.3.2 C Standard Language-Dependent System Support. The issue of "namespace pollution" needs to be understood in this context. See **POSIX Symbols** §B.2.8.2.

B.2.2.3.3 Common Usage C Language-Dependent System Support. The issue of "namespace pollution" needs to be understood in this context. See **POSIX Symbols** §B.2.8.2.

B.2.2.4 Other C Language Related Specifications. The information concerning the use of library functions was adapted from a description in the C Standard. 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 X3J11 for functions with variable arguments; furthermore, parameter type conversion "widening" is subject to different rules in this case.)

Note that the C Standard 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.2.3 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 notoriously hard to define. Some are defined by context in the prose topic descriptions of **General Concepts** §2.4, but most appear in the alphabetical glossary format of **General Terms** §2.3. All technical terms not explicitly defined have definitions in the *IEEE Dictionary*. See **Bibliographic Notes** §B.11.1.

Some definitions must allow extension to cover terms or facilities that are not explicitly mentioned in the standard. For example, the definition of *file* must permit interpretation to include streams, as found in the Eighth Edition. The use of abstract intermediate terms (such as *object* in place 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 the standard; these are marked prefixed with a asterisk (*). Many of them have been specifically excluded from the standard 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 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, the standard allows more granular privilege mechanisms to be defined. For many existing implementations of the UNIX system, the presence of the term *appropriate privileges* in this standard 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.

controlling terminal. The question of which of possibly several special files referring to the terminal is meant is not addressed in the standard.

***cooperating implementation.** This refers to a POSIX implementation that is done in combination with some other set of system specifications. This might be as simple as supporting a POSIX environment concurrently with some specific version of AT&Ts UNIX Operating System, or as complex as providing the POSIX environment with some different vendor's products, such as MS/DOS from Microsoft, VMS from Digital Equipment Company, etc. A cooperating environment

would fall somewhere on the gray scale from hosted implementations to native, depending on the degree of POSIX components that are serviced directly versus those that are converted to correspond with one of the other system's implementations. (Note that the POSIX facilities might be native, and the other system hosted; or both might be native.)

***device number.** The concept is handled in *stat*() §5.6.2 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 **Directory Operations** §5.1.2 and certain constraints on the format of the information returned by those routines are made in **Format of Directory Entries** §5.1.1.

directory entry. Throughout the document, the term *link* is used (about *link*() §5.3.4, for example) in describing the things that point to files from directories.

dot. The symbolic name *dot* is carefully used in the standard to distinguish the working directory filename from period or 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 Application. Therefore such use is considered an extension, the use of which makes an implementation non-conforming. See also **pathname resolution** §B.2.4.

Epoch. Normally, the origin of UNIX system time is 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 this standard 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 not given in this standard.

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. The standard is more concerned about the synchronization of time between applications of astronomically short duration and the Working Group expects these concerns 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 divide is an integer divide (discarding remainder) as in C.

Note also that in Chapter 8, the meaning of *gmtime*(), *localtime*(), and *mktime*() is specified in terms of this expression. However, the C Standard 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 this standard.

FIFO special file. See **pipe** §B.2.3.

file. It is permissible for an implementation-defined file type to be non-readable or non-writable.

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. For now the primary responsibility for truncating filenames containing multibyte characters must reside with the application. The /usr/group subcommittee for internationalization believes that in the future, the responsibility must shift to the kernel. For now, there needs to be time to arrive at a better understanding of the implications of making the kernel responsible for truncation of multibyte file names.

The Working Group felt it would be inadvisable to adopt character level truncation as there is no support in this standard which 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** §B.2.4, and that of a mountable subset of that hierarchy, i.e., a **mounted file system** §B.2.3. The standard 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 **System Databases** §B.9.

***historical implementations.** This refers to previously-existing implementations of programming interfaces and operating systems that are related to the interface specified by the standard, especially to those implementations described by the **Base Documents** §B.1.3. See also **Minimal Changes to Historical Implementations** §B.1.2.8.

***hosted implementation.** This refers to a POSIX implementation that is accomplished through interfaces from the POSIX 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 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 multi-user minicomputers. This wide range would probably share a common POSIX 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 the standard dealing with run-time queries: *sysconf*() §4.8.1 and *pathconf*() §5.7.1.

***incomplete pathname. Absolute pathname** §2.4 has been adequately defined.

job control. In order to understand the job control facilities in POSIX 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 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 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 above interactive interface can be accomplished using the POSIX 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*() §4.3.3 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 *exec*() 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*() §7.2.4 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 job control facilities which allow deviations from the typical interface. Clearing the TOSTOP terminal flag (see **Local Modes** §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 which is not using the job control facilities can be thought of as a large collection of processes which 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 non-interactive function (reading commands from a file). When operating non-interactively, the job control shell will refrain from performing the job control specific actions described above. It will behave as a shell which 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 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 which simultaneously attempt to control mediation of the terminal.) To determine this, the shell retrieves its own process group via *getpgrp*() §4.3.1 and the process group of the current foreground job via *tcgetpgrp*() §7.2.3. 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 doesn't inadvertently stop itself.

Implementing Job Control Applications

Most applications do not need to be aware of job control signals and operations; the "right thing" 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: "I don't know what this signal is, but I know 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 which 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 by Kernighan and Ritchie in *UNIX Programming — Second Edition* in the UNIX Programmer's Manual.

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 that other processes in the job could be children, siblings, or even ancestors.) Another point worth noting is that 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 Applications.

Discussions of the changes can be found in the sections which discuss the specific interfaces. See sections: **Wait for Process Termination** §B.3.2.1, **Terminate a Process** §B.3.2.2, **Signal Names** §B.3.3.1, **Send a Signal to a Process** §B.3.3.2, **Examine and Change Signal Action** §B.3.3.4, **Get Process Group ID** §B.4.3.1, **Set Process Group ID for Job Control** §B.4.3.3, **Terminal Access Control** §B.7.1.1.3, and **Set Foreground Process Group ID** §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 which continues the stopped process. Because a job control shell is allowed, but not required, by the standard, an implementation must provide a mechanism which 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 which 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, the standard 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 which is not job control cognizant are in an orphaned process group and are protected from stopping.

POSIX 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 nothing to 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 Application should not attempt to access the controlling terminal after logout since such access is unreliable. Note that, if an implementation chooses to deny access to a controlling terminal after its controlling process exits, the standard requires a certain type of behavior (see **The Controlling Terminal** §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 that interfaces directly to an operating system kernel addressed in the standard. See also **hosted implementation** §B.2.3 and **cooperating implementation** §B.2.3. A similar concept from the UNIX world is a native UNIX system, which would a be kernel derived from one of AT&T's UNIX products.

***passwd file.** Implementation-defined; see **System Databases** §B.9.

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 which is not orphaned but sent to another process which 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.

pipe. It proved convenient to define a *pipe* as a special case of a *FIFO* even though historically the latter were only introduced in System III and do 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 the standard. See also **filename portability** §B.2.4.

Note that situations where characters beyond the portable filename character set (or simply ASCII) would be used (in a context where the portable filename character set or ASCII is required herein) are expected to be common. Although such a situation renders the use technically non-compliant, 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 this standard. (Making it required would eliminate too many possible systems, as even those systems using ASCII as a base character set extend their character sets for Western Europe, let alone the rest of the world, in different ways.)

Nothing in this standard 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, permitting any extensions in this context defeats the purpose of the standard.

regular file. The standard 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 the standard.

root directory. This definition permits the operation of *chroot*(), even though that function is not in the standard. See also **file hierarchy** §B.2.4.

***root file system.** Implementation-defined.

***root of a file system.** Implementation-defined. See *mount point*.

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 the standard. See **Interface, Not Implementation** §B.1.2.2.

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

B.2.4 General Concepts.

extended security controls. Allowing an implementation to define extended security controls enables the use of this standard in environments which require different or more rigorous security than that provided in the standard. 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.

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*() §5.6.3. An implementation may include other security mechanisms in addition to those specified in the standard, and an access attempt may fail because of those additional mechanisms even though it would succeed according to the rules given in this section. (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.

file hierarchy. Though the file hierarchy is commonly regarded to be a tree, the standard does not define it as such for three reasons:

(1) As noted in the standard, 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.

file permissions. Examples of implementation-defined constraints that may deny access are mandatory labels and access control lists.

filename portability. Traditionally, certain filenames have been reserved. This list includes `core`, `/etc/passwd`, etc. Care should be taken in portable applications to avoid these.

Most historical implementations, including all of those described by the **Base Documents** §B.1.3, prohibit case folding in filenames, i.e., treating upper- and lowercase alphabetic characters as identical. However, some consider case folding desirable

(1) For user convenience.

(2) For ease of implementation of the standard interface as a hosted system on some popular operating systems, which is compatible with the goal of making the standard interface **Broadly Implementable** §B.1.2.7.

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 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 implementors would still have other problems with native file systems, such as short or otherwise constrained filenames or pathnames, not to mention 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 the Working Group is to standardize the common, existing definition (see **Application Oriented** §B.1.2.1) of the UNIX system programming interface, not to change it. Mandating case insensitivity would make all historical implementations non-standard.

(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** §B.1.2.9.

(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 the standard 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:

(1) 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.

(2) Change "though this practice is not recommended:" to "although this practice is strongly discouraged."

Rationale: If case folding must be included in the standard, the wording should be stronger to discourage the practice.

The consensus of the Working Group was in favor of proposal (1). Otherwise, a portable application would have to assume that case folding would occur when it wasn't wanted, but that it wouldn't occur when it was wanted.

file times update. This section 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.

Earlier drafts had not required these times and did not clearly specify the time update process. However, the Working Group felt that the functionality provided was important.

The accuracy of the time update values is intentionally left unspecified so that systems can control the bandwidth of a possible covert channel.

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 the standard permits this behavior.

Other networked systems use the construct `//hostname` for the same purpose, i.e., a double initial slash is used. Because existing applications which create full pathnames by taking a trunk and a relative pathname and making them into a single string separated by / can accidentally create networked

pathnames when the trunk is /, the Working Group considered prohibiting this practice. However, 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 non-leading 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 the standard 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.2.5 Error Numbers. 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 *mkdir*() §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*() §4.7.2 and *ptrace*(). The latter is not in the standard, 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.

The standard requires (in the **Errors** subsections 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 **Error Numbers** §2.5. This section otherwise allows wide latitude to the implementation in handling error reporting.

All references to the term *system call* have been excised from the descriptions of errors in this section.

Some of the **Errors** sections in the standard have two subsections. 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" section. 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:"

has been known to the Working Group as the "optional" section. This latter section has evolved in meaning over time. Originally, 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 section recently has also added conditions associated with optional system behavior, such as job control errors.

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 *perror*() function) when the corresponding errors are detected, but the standard does not require this action.

[EFAULT] Most historical implementations do not catch an error and set *errno* when a bad address is given to the functions *wait*() §3.2.1, *time*() §4.5.1, or *times*() §4.5.2. Some implementations cannot reliably detect a bad address. And most systems that detect bad addresses will do so only for a **system call** §B.2.3, not for a **library routine** §B.2.3.

[EINTR] The standard 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** §B.2.3 has been eliminated from historical versions of this description as being implementation-defined.

[ENOTTY] The symbolic name for this error is derived from a time when device control was done by *ioctl*() §B.7 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 section primarily for consistency with the C Standard.

B.2.6 Primitive System Data Types. Depending on which draft is consulted, a requirement that additional types defined in this section end in "_t" may or may not have been present. The issue of namespace pollution (see **POSIX Symbols** §B.2.8.2) decided this in favor of requiring that "_t" be used. It is difficult to define a type in one header file and use it in another, where that type is not one defined by this standard, without adding symbols to the namespace of the program. To allow implementors to provide their own types, programs which use **<sys/types.h>** are required by this standard 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 this standard, the need for additional types is compelling.

The types such as *ushort* and *ulong*, which are in common usage, are not defined in this standard (although *ushort_t* would be permitted as an extension). They can be added to **<sys/types.h>** using a feature test macro (see **POSIX Symbols** §2.8.2). 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 **C Language Definitions** §2.8.

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 non-arithmetic (such as a structure) and provided a *samefile*() function for comparison.

gid_t Some implementations had separated *gid_t* from *uid_t* before this standard was completed. It would be a burden 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. 4.3BSD uses *unsigned short* and the *SVID* uses *ushort*, which is the same thing. Historically, only the low-order sixteen bits are significant.

nlink_t This type was introduced in place of *short* for *st_nlink* (see **<sys/stat.h>** §5.6.1) in response to an objection that *short* was too small.

off_t This type is used only in *lseek*() §6.5.3, *fcntl*() §6.5.2, and **<sys/stat.h>** §5.6.1. Many implementations would have difficulties if it were defined as anything other than *long*. The

Working Group realizes that requiring an integral type limits the capabilities of *lseek*() to four gigabytes. See *lread*() §B.6.4. Also, the C Standard supplies routines that use larger types: see *fgetpos*() §B.6.5.3 and *fsetpos*() §B.6.5.3.

pid_t There has been a lot of debate about the inclusion of this symbol. Much of 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" that are produced by calls such as *fork*(), and used by calls such as *waitpid*() or *kill*(), and which are not otherwise analyzed. (Except that sign is used as a flag for certain operations.)

The concept of {PID_MAX} interacted with this. Treating PIDs as an opaque type both removes the requirement for {PID_MAX} and allows system to be more flexible in providing PIDs that span a large range of values, or a small one.

Since in the general case the values in *uid_t*, *gid_t*, and *pid_t* will just be numbers, and will be potentially both large in magnitude and sparse, applications which are based on arrays of objects of this type are unlikely to be fully portable in any case. Solutions which treat them as magic cookies will be portable.

{CHILD_MAX} precludes the possibility of a "toy" implementation where there would only be one process.

uid_t Before the addition of this type, the data types used to represent these values varied throughout the standard. The **<sys/stat.h>** §5.6.1 header defined these values as type *short*, the **<passwd.h>** file (now **<pwd.h>** §9.2.2 and **<grp.h>** §9.2.1) used an *int* and *getuid*() §4.2.1 returned an *int*. In response to a strong objection to the inconsistent definitions, the Working Group decided to switch all the types 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 main problem associated with this change is a concern about 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* §9.2.2 structure. Consequently, implementations with an overriding concern for object compatibility can pad the structure back to its current size. For that reason, this problem wasn't considered critical enough to warrant the addition of a separate type to the standard.

The types *uid_t* and *gid_t* are magic cookies. There is no {UID_MAX} defined by the standard, 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.2.7 Environment Description.

LC_* The description of the environment variable names starting with the characters "LC_" acknowledges the fact that the interfaces presented in the draft are not complete and may be extended as new international functionality is required. In the X3J11 draft proposal, names preceded by "LC_" are reserved in the name space for future categories.

To avoid name clashes, new categories and environments variables will be 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 letter, 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 non-conforming. The C shell and the Korn shell conform to the standard on this point. The usual name of **dot** §2.3 may also be used to refer to the current working directory.

TZ See **Extensions to Time Functions** §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 **portable filename character set** §B.2.3.)

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 this standard, 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.2.8 C Language Definitions. The construct **<name.h>** for headers is also taken from the C Standard.

B.2.8.1 Symbols From The C Standard. The reservation of identifiers is taken directly from the C Standard. It is quoted because it needs to be part of this standard, and because by quoting, if the C Standard does change, the requirement here will not change, which is intended. The reservation of other namespaces is particularly for **<errno.h>**.

These identifiers may be used by implementations, particularly for feature test macros. Care needs to be taken to assure that implementations do not use feature test macro names that might be reasonably used by a standard.

Headers being included more than once is a reasonably common practice, and should be carried forward from the C Standard. 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 and within this standard: *time_t* should be in **<sys/types.h>**, and the C Standard mandates that it be in **<time.h>**. This standard requires using **<sys/types.h>** with **<time.h>** because of the common usage environment.

B.2.8.2 POSIX Symbols. This section addresses the issue of "namespace pollution." The C Standard requires that the namespace beyond what it reserves not be altered except by explicit action of the application writer. This section defines the actions to add the POSIX symbols for those headers where both the C Standard and POSIX need to define symbols. Where there are non-overlapping uses of headers, of course there is no problem.

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 way because it is expected that future additions to this standard 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 sections: 2.8.2.1 covers the case of the C Standard conformant systems, where the requirements of the C Standard are that unless specifically requested you 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 tricky part is common usage. In the C Standard 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 and POSIX. (It is up to X3J11 to determine if they want a similar macro to limit the features to just the C Standard; 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 is the default they will have to change (unless they are already C Standard 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: you won't get implementation defined symbols unless you ask for them. (But if you don't use _POSIX_SOURCE you may or may not; you get the implementation default, which is probably backwards compatible.)

Because members of structures and unions are a limited namespace, it is not an issue of namespace pollution for an implementor to add fields to a structure or union, and that special case is excepted.

There is also a special case for names of predefined forms. This is to permit adding symbols (typedefs) of the form *xxx_t* to **<sys/types.h>**, which can be used by structures or unions which appear in other header files. It is extremely difficult to have these types appear only in the context where they are needed (for example, in the extra fields of a *stat* or *dir* structure) without this exception.

Lastly, the standard is silent about whether additional symbols can be defined in headers, as long as they are under control of _POSIX_SOURCE, with the exception discussed above. This permits implementors to make a best judgement decision on such issues.

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 following, derived from common porting bases and "pseudo-standards," are suggested. (The "owners" of these various bases are generally fairly clear, and if these symbols are inappropriate or inaccurate they could suggest others, possibly to be included in later editions of this document.)

Symbol	Description
_V7	Version 7
_BSD	General BSD systems.
_BSD4_2	4.2BSD
_BSD4_3	4.3BSD
_SYSIII	AT&T System III
_SYSV	AT&T System V.1 V.2
_SYSV3	AT&T 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 pseudo-standard 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 this standard.

The environment for compilation has traditionally been fairly portable in historical systems, but during the transition to the C Standard there will be confusion about how to specify that a C Standard 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 which are sufficiently similar to traditional UNIX systems for this to make sense (e.g. those that will probably conform to P1003.2), it is suggested that if a compilation line of the form

```
cc -D__STDH__ ...
```

is provided, that the system provide an environment that is conformant with the C Standard, at least with respect to namespace. (It is not expected that this will be required in the future or that it will be formally standardized. It is suggested as a convention during the transition.)

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.

The standard 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 doesn't 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.2.8.3 Headers and Function Prototypes.

B.2.9 Numerical Limits. This section has been completely rewritten since the Trial Use Standard, in order to clarify the scope and mutability of several classes of limits.

B.2.9.1 C Language Limits. See also **C Language Definitions** §2.8 and **POSIX and the C Standard** §B.1.4.

{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.

No limits are given in **<limits.h>** for floating point values because none of the functions in the standard proper use floating point values and all the functions that do that are imported from the C Standard by **Referenced C Language Routines** §8.1 defined in the C Standard, as are the limits that apply to the floating point values associated with them.

Though limits to the addresses to system calls were proposed, it is not clear how to implement them for the range of systems being considered and, lacking a complete proposal, the Working Group determined not to attempt this at this time. Limits regarding hardware register characteristics were similarly proposed and not attempted.

B.2.9.2 Minimum Values. There has been a lot of 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 **Run-Time Invariant Values (Possibly Indeterminate)** §2.9.4 where an indeterminate value will leave the programmer with no symbol to fall back upon.

By providing explicit symbols for the minima (from the implementor's point of view, or maxima from the the application's point of view) this helps clear up a lot of possible confusion. Symbols are still provided for the actual value, and it is expected that many applications will take advantage of the larger values, but they need not do so unless it is to their advantage. Where the values in this section are adequate for the application, it should use them. These are given symbolically both because it makes the standard easier to understand and because the values of these symbols could change between revisions of the standard. Arguments to "good programming practice" also apply.

B.2.9.3 Run-Time Increasable Values. The heading of the rightmost 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 Application can depend on finding when translated to execute on that implementation. A Conforming POSIX Application Using Extensions must function correctly even if the value given in **<limits.h>** is the minimum that is specified in the standard. (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 the standard. An implementation that cannot meet this requirement (a "toy implementation") cannot be a conforming implementation.

B.2.9.4 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.2.9.5 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} in early drafts. Application writers should use {MAX_INPUT} primarily as an indication of the number of bytes that can be written a single unit by one Conforming POSIX 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 **Input Modes** §7.1.2.2).

Understanding that the limit exists and knowing its magnitude is important to making certain classes of applications work correctly. It seems unlikely that it would be used in an application, but its presence makes the standard clearer.

{PATH_MAX}

A Conforming POSIX Application or Conforming POSIX 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 counter-productive to include an artificially small fixed value in **<limits.h>** in such cases.

B.2.10 Symbolic Constants.

B.2.10.1 Symbolic Constants for the *access*() Function.

B.2.10.2 Symbolic Constants for the *lseek*() Function.

B.2.10.3 Compile-Time Symbolic Constants for Portability Specifications. Related material appeared in an appendix of the Trial Use Standard. The purpose there was 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 (see **Verification Testing** §A.2.4) for the standard. The constants given here were originally proposed for a separate file, **<posix.h>**, but the Working Group decided that they should appear in **<unistd.h>** along with other symbolic constants.

B.2.10.4 Execution-Time Symbolic Constants for Portability Specifications. Without the addition of {_POSIX_NO_TRUNC} and {_PC_NO_TRUNC} to this list, the standard 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. The Working Group considered enumerating all characteristics of a process that the standard defines, 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 existing 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*() §3.2.1 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 the present standard, although they are currently being addressed by the 1003.2 Working Group.

B.3.1.1 Process Creation. Many existing 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 the child, because the *fork*() code clears the child's set of pending signals. It is not the intention of this standard to 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 the standard and pass available verification suites. The Trial Use Standard mentioned this behavior in a Note that was not intended to be part of the standard, but gave the appearance of permitting the behavior; this Note has been removed. 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, from that view the set would be initialized as empty and such signals would have arrived after the *fork*(). See also **Signal Generation and Delivery** §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 as good a fix as possible. 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] in not appropriate 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 existing 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* specifies it and it should be reserved for the error condition specified there. The condition is not applicable on many implementations.

B.3.1.2 Execute a File. The use of ellipses (...) in the **Synopsis** entries for *execl*(), *execle*(), and *execlp*() is not intended to be strict C language syntax, but to represent a concept that cannot be expressed fully in C.

The Trial Use Standard 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. In fact, traditional 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 and subsequently removed from this standard as well. Note however that the wording of this standard, in particular the use of the word "should," requires a Strictly Conforming POSIX Application (see **Language-Dependent Services for the C Programming Language** §2.2.3) to pass at least one argument to the *exec* function, thus guaranteeing than *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*.

Note that the requirement on a Strictly Conforming POSIX Application also states that the value passed as the first argument be a filename associated with the process being started. Although some existing applications pass a pathname rather than a filename in some circumstances, a filename is more generally useful, since the common usage of *argv*[0] is in printing diagnostics. In some cases the filename passed is not the actual filename of the file; for example many implementations of the `login` utility use a convention of prefixing a hyphen (–) to the actual filename, which indicates to the command interpreter being invoked that it is a "login shell."

Some systems can *exec* shell scripts. This functionality is outside the scope of this standard, 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 P1003.2 standard. However, it is important that this standard neither require nor preclude any reasonable implementation of this behavior. In particular, the description of the [ENOEXEC] error is intended to permit implementations discretion on whether to give this error for shell scripts.

One common existing implementation is that the *execl*(), *execv*(), *execle*(), and *execve*() functions return an [ENOEXEC] error for any file not recognizable as an 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 existing implementations handle shell scripts is by recognizing the first two bytes of the file as ASCII `#!` 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 specifies *main*() to be invoked with two arguments, so implementations must support applications

written this way. Since this standard defines the global variable *environ*, which is also provided by existing implementations and can be used anywhere *envp* could be used, there is no functional need for the *envp* argument. Furthermore, application writers are usually better off using the *getenv()* function 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.

The standard 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 traditional 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, 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.

[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 existing systems return [EFAULT] rather than [ENOEXEC] when the new process image file is corrupted. They are non-conforming.

[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. The standard neither requires nor prohibits this behavior.

Other systems (such as System V) may return [EINTR] from *exec*. This is not addressed by the standard, 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. The Trial Use Standard 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 the standard distinguished

three types of termination: normal termination (as in the current standard), "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 standard was only saying when they 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 the standard.

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 the standard 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 on 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*() §3.1.1 call (perhaps followed by an *exec* §3.1.2 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 the standard 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. One is to support job control (see **Signals** §B.3.3). The second is to permit a non-blocking version of the *wait*() function. The third is to permit a library routine, such as *system*() or *pclose*(), to wait for its children without interfering with other terminated children that have not been waited for.

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 process in that implementation: processes being traced via the *ptrace*() debugging facility, and (untraced) processes stopped by job control signals. Since *ptrace*() is not part of this standard, only the second type is relevant. The working group chose to retain the

name WUNTRACED 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, consider the problem that the IEEE Std 1003.2 working group addressed:

```
stream = popen("/bin/true");
(void) system("sleep 100");
(void) pclose(stream);
```

On all traditional implementations, the final *pclose*() will fail to reap the wait status of the *popen*().

The status values are retrieved by macros rather than being given as specific bit encodings as given 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. The standard does require that a status value of zero correspond 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 specification 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 the standard 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 traditional implementations is the *ptrace*() function. It is called by a child process and causes that child to stop and return 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 function *_exit*() is defined here instead of *exit*() because the C Standard defines the latter to have certain characteristics that are beyond the scope of the present standard, specifically the flushing of buffers on open files and the use of *atexit*(). See **The C Language and X3J11** §B.1.2.4. There are several public domain implementations of *atexit*() which 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 section 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 **Process Termination** §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 section. Implementations in languages other than C are outside the scope of the present standard, however.

As required by X3J11, using `return` from *main*() §3.1.2 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 to be zero in **Referenced C Language Routines** §8.1) for the argument *status* conventionally indicates successful termination. This corresponds to the specification for *exit*() in the C Standard. The convention is followed by utilities such as `make` and various shells, which interpret a zero status from a child process as success. For this reason, applications should not call *exit*(0) or *_exit*(0) when they terminate unsuccessfully, for example in signal-catching functions.

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

The sending of a SIGHUP to the foreground process group when a controlling process terminates corresponds to somewhat different existing 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 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 receive similar treatment at that time. In POSIX, the signals will be sent to the entire process group at the same time. Also, in POSIX 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.

B.3.3 Signals. Signals, as defined in the Trial Use Standard, and in Version 7, System III, the *1984 /usr/group Standard*, and System V (except very recent releases), have shortcomings which make them unreliable for many application uses. Several objections were voiced against the Trial Use Standard because of this. Therefore a new signal mechanism, based very closely on the one of 4.2BSD and 4.3BSD, was added to the standard. With the exception of one feature (see item 4 below and also *sigpending*() §3.3.6), it is possible to implement the POSIX 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 the standard.

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 thirty-two). The new standard instead uses a defined type for signal masks. Because of this change, the interface is significantly different than 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. The standard (in *sigaction*() §3.3.4) does not require restart of functions, because it was not clear that the semantics of system call restart in any existing 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 the standard because these functions are not portable, and no sufficiently portable and useful equivalent has been identified. See also **Non-local Jumps** §8.4.

(4) **Pending signals.** The *sigpending*() §3.3.6 function is the sole new signal operation introduced in the standard. It was requested by some

members of the Working Group and was seen as a simple and useful feature.

The Working Group considered 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 traditional implementations of the *signal*() mechanism defined by the C Standard. 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 does not require this behavior, but does explicitly permit it, and most existing 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 no longer requires this unreliable behavior, many existing implementations, including System V, will reset the signal action to SIG_DFL. For this reason, the Working Group strongly recommends that the *signal*() function not be used by POSIX 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 traditional 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 Working Group decided to neither require nor prohibit the queueing of signals. See **Signal Generation and Delivery** §3.3.1.2. It is expected that a future Realtime Extension to this standard (see **Realtime Extensions** §A.2.5) 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 the document 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 the standard 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 the standard because they do indicate three relatively well categorized conditions. They are all defined by the C Standard, and thus would have to be defined by any system with a C Standard binding, even if not explicitly included in this standard.

There is very little if anything that a Conforming POSIX 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 which have terminated or, as in 4.2BSD, stopped. See also **Examine and Change Signal Action** §B.3.3.4.

Some implementations, including System V, have a signal named SIGCLD which is similar to SIGCHLD in 4.2BSD. It is the intention of the standard to permit implementations to have a single signal with both names. The standard carefully specifies ways in which portable applications can avoid the semantic differences between the two different implementations. The name SIGCHLD was chosen for the standard because most current application usage of it can remain unchanged in conforming applications. SIGCLD in System V has more cases of semantics which the standard 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 the standard requires that symbolic names always be defined (with the exception of certain names in **<limits.h>** §2.9 and **<unistd.h>** §2.10), 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*()

§3.3.2. 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 non-portable libraries to use non-standard signals to avoid conflicts with use of standard signals by portable libraries.

The Trial Use Standard distinguished which signals do or don't cause "implementation-defined actions" as part of abnormal termination as their default action on delivery. See **Process Termination** §B.3.2..

There is no portable way for an application to catch or ignore non-standard 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 the standard. 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 section are not used consistently in documentation of existing systems. Each signal can be considered to have a lifetime beginning with *generation* and ending with *delivery*. The Working Group considered defining *delivery* to exclude ignored signals, but the current definitions were chosen as more uniform.

Implementations should deliver unblocked signals as soon after they are generated as possible. However, it is difficult for the standard to make specific requirements about this, beyond those in *kill*() §3.3.2 and *sigprocmask*() §3.3.5. 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 the standard to specify behavior that would invalidate any of the existing implementations.

There is one case in some existing 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 the standard. Some earlier drafts of the standard explicitly stated this, but these statements were redundant due to the requirement that

functions defined by the standard not change attributes of processes defined by the standard except as explicitly stated (see **Process Primitives** §3).

The standard specifically states that the order in which multiple, simultaneously pending signals are delivered is unspecified. This order has not been explicitly specified in traditional 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 people 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 in widespread use, the Working Group did not wish to standardize, nor necessarily endorse, this behavior. 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. The reason is that, 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. At one time the working group considered disallowing catching SIGCONT as well as 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 the standard mentioned SIGCONT as a second exception to the rule that signals are not delivered to stopped processes until continued. Because the standard 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.

Traditional 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. The standard requires this for the sake of consistency and also for completeness, since the only signal

this applies to is SIGCHLD and the standard 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*() §3.2.1. 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 the standard 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 effective, but probably too harsh, as SIGKILL is not catchable. Another possible choice is SIGHUP. 4.3BSD also does this for orphaned processes (processes whose parent has terminated) rather than members of orphaned process groups; this is less desirable because job control shells manage process groups. The standard 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 **Terminal Access**

Control §B.7.1.1.4.) The Working Group considered a similar restriction on the generation of SIGTSTP, but that would be unnecessary and more difficult to implement due to its asynchronous nature.

Although the standard requires that signal-catching functions be called with only one argument, there is nothing to prevent conforming implementations from extending the standard to pass additional arguments, as long as Strictly Conforming POSIX Applications continue to compile and execute correctly. Most traditional 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 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 allows implementations to use a different parameter passing mechanism for variable parameter lists than for non-variable 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 to be portable. This is in conflict with the goal of **Minimal Changes to Existing Application Code** §B.1.2.9.

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 non-zero 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 *_exit*() §B.3.2.2).

The behavior of *unsafe* functions, as defined by this section, is undefined when they are invoked from signal-catching functions in certain circumstances. The behavior of reentrant functions, as defined by this section, is as specified by the standard, 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*() §3.4.3 and *sleep*() §B.3.4.3) and interactions among multiple handles for a file description (see **Interactions of Other FILE-Type C Functions** §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 non-reentrant function calls, applications should protect calls to these functions either by blocking the appropriate signals or through the use of some programmatic semaphore. The standard 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. Naturally, the same principles apply to the reentrancy of application routines and asynchronous data access.

Note that *longjmp*() and *siglongjmp*() are not in the list of reentrant functions. This is because the code executing after the *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 non-reentrant manner. Because any combination of different functions using a common data structure can cause reentrancy problems, the standard 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. In the Trial Use Standard the effect of signals interrupting functions was described under signals. Because of the various different cases, they are now described under each of the interruptible 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 traditional implementations of many functions defined by this standard 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) which are interruptible. This permits applications to respond to interactive signals or to set timeouts on calls to most such function with *alarm*(). Therefore implementations should generally make such functions (including ones defined as extensions) interruptible.

Functions not mentioned explicitly as interruptible may be 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 the standard agree with System V. Specifically, a *setuid* 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. The statement that "the implementation may still permit" a *setuid* application to receive such signals is

intended as a warning to application writers about this fact; the word *may* should be interpreted as describing a possible scenario, not as a specification of any optional semantics.

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, the Working Group felt that this behavior should not be standardized, 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 existing 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 which, 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 non-super-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 the standard.

There was strong sentiment in the Working Group 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, traditional implementations that provide only the *signal*() interface only make the weaker guarantee in the standard, 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 Working Group 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*() §3.3.5.

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 which 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 non-existence of a process from being used as a covert channel, such processes should appear non-existent 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 this standard'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*() §3.2.1).

Various individuals have indicated that the name *kill*() is misleading, since the function is not always intended to cause process termination. However, the name is common to all traditional implementations, and any change would be in conflict with the goal of **Minimal Changes to Existing Application Code** §B.1.2.9.

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 this standard. Although it is outside the scope of this standard 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 the standard 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 the distinction, it does not make sense 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 the standard 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 of invoking the signal-catching routine when they stop.

Some existing implementations also define other mechanisms for stopping processes, such as the *ptrace*() function. These implementations usually do not generate a SIGCHLD signal when processes stop due to this mechanism, however that is beyond the scope of this standard.

The standard 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*().

The standard requires that *sigaction*() properly save and restore a signal action set up by the C Standard *signal*() function. However, there is no guarantee that the reverse be 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 is the intention of the Working Group that *signal*() should be implementable as a library routine using *sigaction*().

B.3.3.5 Examine and Change Blocked Signals. Note that when a process's signal mask is changed in a signal-catching function that is installed by *sigaction*(), that the restoration of the signal mask on return from the signal-catching function overrides that change (see *sigaction*() §3.3.4). If the signal-catching function was installed with *signal*() it is unspecified whether this occurs.

See *kill*() §B.3.3.2 for a discussion of the requirement on delivery of signals.

B.3.3.6 Examine Pending Signals.

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 traditional 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 early, up to 1/{CLK_TCK} seconds 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 "real-time 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 Application cannot pass a value greater than the minimum guaranteed value for {UINT_MAX}, which the C Standard sets as 65 535, and any application passing a larger value is restricting its portability. The Working Group considered using a different type, but existing 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*() §3.4.3 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. The standard permits either approach.

In order to comply with the wording of the introduction to Chapter 3, that no primitive shall change a process attribute unless explicitly described by the standard, 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.

Traditional implementations often implement the SIGALRM-based version using *alarm*() and *pause*(). One such implementation is prone to infinite hangs as described in *pause*() §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 implementation. These are the cases mentioned in the standard 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 **Schedule Alarm** §B.3.4.1.)

Because *sleep*() can be implemented using *alarm*(), the discussion about alarms occurring early under *alarm*() §B.3.4.1 apply 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 Application cannot pass a value greater than the minimum guaranteed value for {UINT_MAX}, which the C Standard sets as 65 535, and any application passing a larger value is restricting its portability. The Working Group considered using a different type, but existing 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.

B.4.2 User Identification.

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

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* §3.1.2 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*() §B.4.2.3 is a privileged operation and therefore is not covered by this standard.

The effective group ID may appear in the array returned by *getgroups*(), or it may be returned only by *getegid*(). This is a warning to programmers that duplication may exist, and that one needs to call *getegid*() to be sure of getting all of the information. Various implementation and administrative sequences (how did you get into these sets of groups, etc.) will cause these to vary.

B.4.2.4 Get User Name. L_cuserid 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.4.3 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 the standard.

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 adds the concept of a session which 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 session containing a single POSIX process group. Note that this function requires that the calling process not be a process group leader. The usual way to ensure this is true is to create a new process with *fork*() and have it call *setsid*(). The *fork*() function guarantees that the process ID of the new process does not match any existing process group ID.

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 **job control** §B.2.3.

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*(). The Working Group considered adding a new version of *fork*() with this same synchronization property, but 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 doesn't 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 non-obvious use of *setpgid*() is to allow a job control shell to return itself to its original process group (the one in effect when the job control shell was executed). A job control shell does this before returning control back to its parent when it is terminating or suspending itself as a way of restoring its job control "state" back to what its parent would expect. (Note that the original process group of the job control shell typically matches the process group of its parent, but this is not necessarily always the case.) See also *tcsetpgrp*() §B.7.1.7.

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 this interface standard implemented in the operating system. An application implementor should instead depend on {_POSIX_VERSION} and related constants defined in **Symbolic Constants** §2.10.

The standard 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 existing 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*() §4.5.1 function returns a value in seconds (type *time_t*) while *times*() §4.5.2 returns a set of values in {CLK_TCK}ths of a second (type *clock_t*).

Some historical implementations, such as 4.3BSD, have mechanisms capable of returning more precise times (see *gettimeofday*() §B.4.5.1). A generalized timing scheme to unify these various timing mechanisms has been proposed but not adopted in this standard; see **Realtime Extensions** §A.2.5.

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. The Working Group chose not to try to fix this. But they did require the use of *time_t* 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.

Many historical implementations (including Version 7) and the *1984 /usr/group Standard* use *long* instead of *time_t*. The present standard uses the latter type in order to agree with the C Standard.

4.3BSD includes *time*() only as an interface to the more flexible *gettimeofday*() function.

B.4.5.2 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 **Schedule Alarm** §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 {CLK_TCK} is 60, or less than a year if it is 100. There are individual systems that run continuously for longer than that. The standard 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 a supplement to this standard because uses from within an application have been identified since the decision was made.

It was proposed that this function is properly part of Chapter 8. It is an extension to a function in Standard C. Because this function should be available from any language, not just C, it appears here, to separate it from the material in Chapter 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 section was added in response to requirements of application developers, and particularly the X/Open system vendors. It is closely related to **Configurable Pathname Variables** §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 the standard, 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 the standard.

(2) International applications vendors occasionally require knowledge of the {CLK_TCK} value. Without the facilities of this section, they would be required to either distribute their applications partially in source form or to have 50 Hertz and 60 Hertz versions for the various countries they do business in.

It is the understanding 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, it was felt by the Working Group 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 very heated arguments that accompanied 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 waters. 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. The standard 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 **Configurable Pathname Variables** §B.5.7.

B.4.8.1 Get Configurable System Variables. Note that all values returned must be expressible as integers. The Working Group considered using string values, 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 isn't even defined, in this case.

{CLK_TCK} is not defined in languages other than C, so {_SC_CLK_TCK} is required for those languages. Because {CLK_TCK} has been in a state of change in the C Standard, there is a possibility of redundancy between that standard and POSIX because the number of ticks per second can be accessed both via {CLK_TCK} and via *sysconf*(). {CLK_TCK} is the preferred mechanism to access this value if it reflects the possibility of variation (as permitted by the C Standard at this writing). In fact, an attractive way to implement {CLK_TCK} is using *sysconf*(). To assure that there is at least one way to access the value, *sysconf*() continues to support access to the value. The specification assures that the redundancy won't lead to different values.

B.5 Files and Directories. See **pathname resolution** §2.4.

The wording regarding the group of a newly-created regular file, directory, or FIFO in *open*() §5.3.1, *mkdir*() §5.4.1, *mkfifo*() §5.4.2, 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*() §5.6.5 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*() §6.4.1 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 the standard eliminate that bug, as well as hiding differences in implementations of directories or file systems.

The directory access functions as described in an Appendix of the POSIX Trial Use Standard were derived from 4.2BSD, were adopted in System V Release 3 and are in *SVID* 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** §2.3) in this context, the *d_ino* field does not appear in the present standard. Also, programs that want this information can get it with *stat*() §5.6.2.

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 the standard 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 the standard does not completely define what is in the structure, so it could be different on some implementations from *struct direct*.

The name of a character array of an unspecified size should not be used as an *lvalue*. Use of

```
sizeof (d_name)
```

is incorrect; use

```
strlen (d_name)
```

instead.

This description of the *d_name* element was changed because the previous version gave the impression that the character array *d_name* was of 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 Working Group 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, the standard does not mandate that the directory stream be implemented using file descriptors. Wording was added to the description of *opendir*() to clarify that if a file descriptor is used for the directory stream it is mandatory that *closedir*() deallocate the file descriptor.

The returned value of *readdir*() merely *represents* a directory entry. No equivalence should be inferred.

Since the structure and buffer allocation, if any, for directory operations are defined by the implementation, the standard 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 which 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 non-zero value in case (1), 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, the Working Group decided not to include *seekdir*() and *telldir*() in the standard.

An error or signal indicating that a directory has changed while open was considered but rejected.

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 cannot supply a *buf* with *size* {{PATH_MAX} + 1} in general.

Having the routine 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 the present standard. Finally, *getcwd*() is taken from the *SVID*, where it has the two arguments used in the standard.

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, [EACCESS], 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 the standard 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 anyway 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.5.3 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 the standard, the flags allowed in *oflag* are not mutually exclusive and any number of them may be used simultaneously.

Some implementations permit opening FIFOs with O_RDWR. Since FIFOs could be implemented in other ways, and since two file descriptors can be used to the same effect, this possibility is left as undefined.

See *getgroups*() §B.4.2.3 about the group of a newly-created file.

The use of *open*() §5.3.1 to create a regular file is preferable to the use of *creat*() §5.3.2 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). 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. The standard does not specify how a controlling terminal is acquired, but it allows an implementation to provide this on *open*() if the O_NOCTTY flag is not set and other conditions specified in **The Controlling Terminal** §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 and revisions would be wise to 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 **Input and Output Primitives** §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* §B.2.6.

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 the standard.

B.5.3.4 Link to a File. See **directory entry** §B.2.3.

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. This standard 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.

B.5.4 Special File Creation.

B.5.4.1 Make a Directory. See *mode_t* §B.2.6.

The *mkdir*() function originated in 4.2BSD and was added to System V in Release 3.0, following the Trial Use Standard.

4.3BSD detects [ENAMETOOLONG].

See *getgroups*() §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 existing implementations of *mknod*(). The latter function was included in the *1984 /usr/group Standard*, but only for use in creating FIFO special files. The *mknod*() function was excluded from POSIX as implementation-defined and replaced by *mkdir*() §5.4.1 and *mkfifo*() §5.4.2.

See **getgroups** §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* was published, it contained [EEXIST] instead. When AT&T adopted these functions into System V, they used the /usr/group Standard as their reference.

Therefore, several existing applications and implementations support/use both forms and the Working Group could not agree 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.

When this function was added to System V (in Release 3.0) it used [ENOENT] where the standard uses [ENAMETOOLONG]. Volume 3 of the *SVID*, page 129, states: "FUTURE DIRECTION: To conform with the IEEE POSIX standard, when it is adopted as a full-use standard, the value of *errno* indicating that ..."

B.5.5.1 Remove Directory Entries. Unlinking a directory is restricted to the super-user in many historical implementations for reasons given in *link*() §B.5.3.4. But see *rename*() §B.5.5.3.

The meaning of [EBUSY] in traditional implementations is "mount point busy." Since this standard 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 **File Removal** §B.5.5 and **[EBUSY]** §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. 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*() §5.3.4 and *unlink*() §5.5.1 with directories, thus constraining links to directories to those made by *mkdir*() §5.4.1.

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 **File Removal** §B.5.5 and [EBUSY] in **Remove Directory Entries** §B.5.5.1.

B.5.6 File Characteristics. The function *ustat*(), which appeared in the *1984 /usr/group Standard* and is still in the *SVID*, was removed from the present standard before Trial Use because it was:

(1) 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.

(2) Not required. The only known program that uses it is the text editor `ed`.

It was also not readily extensible to networked systems.

B.5.6.1 File Characteristics: Header and Data Structure. See **Primitive System Data Types** §B.2.6.

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*() §5.3.1, 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 conforming systems should not update the time entry fields for functions listed in the standard 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 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. Some Working Group discussions centered around inadequacies in the *access*() function led to the creation of an *eaccess*() function because:

(1) Historical implementations of *access*() don't 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.

After *eaccess*() was reviewed, the Working Group found that it still didn't resolve problem (1), so the standard 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 there, rather than creating a new function that wouldn't be as reliable. Therefore, *eaccess*() was taken back out of the standard.

Secure implementations will probably need an extended *access*()-like function, but the Working Group did not have enough of the requirements to define

it yet. This could be proposed as an extension to the Full Use Standard. See **Trusted System Extensions** §A.2.7.

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. The standard 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 doesn't 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 which use the S_ISUID bit to indicate some other function (for example, mandatory record locking) on non-executable files need not clear this bit on writing. They should clear the bit for executable files and any other cases where the bit grants special powers to processes that change the file contents. Similar comments apply to the S_ISGID bit.

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 which 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. The standard 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 non-privileged processes, the S_ISUID and S_ISGID bits be cleared on regular files but only *may* be cleared on non-regular 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 the standard does not require that these bits be cleared except on regular files. Note that 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*() §5.6.2 to retrieve the file's *st_atime* and *st_mtime* §5.6.1.3 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 **Configurable System Variables** §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, and; a Berkeley "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 this standard. The *pathconf*() function allows the application to take advantage of the most liberal file system available at run-time. In many Berkeley-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, Berkeley 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 be able to 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 are not mounted on. The value may change when crossing a mount point, so applications that need to know should check for each directory. (Note that 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 **Run-Time Invariant Values (Possibly Indeterminate)** §2.9.4 or **Pathname Variable Values** §2.9.5 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. Rationale for the Change from O_NDELAY to O_NONBLOCK.

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*() §6.4.1 or *write*() §6.4.2 call which 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* includes an interface with some features from both AT&T 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.

The problem with the *1984 /usr/group Standard* is that it does not allow compatibility with existing applications. An implementation cannot both conform to this standard and support applications written for existing AT&T or BSD systems. This was the cause of at least one objection during the trial-use balloting. Several changes have been considered, either at that time or more recently, to address this issue. These include:

(1) No change (from *1984 /usr/group Standard*);
(2) Changing to System III/V semantics;
(3) Changing to BSD semantics;
(4) Broadening the standard 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.

The consensus of the Working Group was that (5) is the best alternative. 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 AT&T or BSD systems (but the standard does not require or even suggest that this be done). It also allows a Conforming POSIX 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 AT&T and BSD applications that use this facility to be modified in order to strictly conform to the standard. 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 **Data Definitions for File Control Operations** §B.5.6.1.

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 this standard 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*() §5.3.1.

See also **Write to a Pipe** §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 this standard primarily for historical reasons, since many existing applications use them.

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*(). The standard only intends to permit such behavior by specifying the [EINTR] error case.

B.6.4 Input and Output. Whether the return values of, and *nbyte* arguments to, *read*() §6.4.1 and *write*() §6.4.2 should be signed or unsigned was a chronic source of controversy. On machines where type *int* is of sixteen bits, only 32 767 bytes may be transferred on one function call. If *nbyte* were unsigned, it would be convenient for the return value to be of the same type. But if the returned value were unsigned, it would be necessary to compare it to ((*unsigned*)–1) in order to detect an error. Although a definition such as IO_ERR could be provided to simplify code, still many existing applications would not conform.

The Working Group decided to make *nbyte* unsigned, with the results of use of values greater than {INT_MAX} (often 32 767) being made implementation-defined. However, the return value was left signed to avoid the error-detection problem. It is still possible to compare the return value directly with *nbyte*, since the C Standard specifies that the comparison will be done unsigned.

Use of the type *long* was considered in order to avoid the sixteen bit problem, but not adopted.

New functions like *read*() and *write*() called *lread*() and *lwrite*() and differing only in that their *nbyte* argument and return values would be of type *off_t* §2.6 were proposed but rejected. The Working Group is not necessarily against the creation of *lread*() and *lwrite*() calls, but was unable to clearly identify the need given the above. It was also noted that C has similar constraints parallel to those mentioned above, and that the type of *sizeof* is not necessarily *long* (where the largest object cannot exceed `sizeof(char[INT_MAX])`.

The standard requires that no action be taken when *nbyte* is zero. This is not intended to take precedence over detection of errors (such as bad buffer pointers or file descriptors). This is consistent with the rest of the standard, 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, the Working Group determined that this should correspond with similar C Language facilities, and that is beyond the scope of the 1003.1 effort. The concept was suggested to X3J11 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 the standard. 4.3BSD semantics can be provided by extensions to this standard.

The standard 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. This unhappily does make writing a Conforming POSIX 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.

B.6.4.1 Read from a File. The standard 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.

References to actions taken on an "unrecoverable error" have been removed. It is considered beyond the scope of this standard to describe what happens in the case of hardware errors.

B.6.4.2 Write to a File. An attempt to write to a pipe or FIFO has several major characteristics:

Atomic/non-atomic

> A write is atomic if the whole amount written in one operation is not interleaved with data from any other process. This is useful when there are multiple writers sending data to a single reader. Applications need to know how large a write request can be expected to be performed atomically. This maximum is called {PIPE_BUF}. The standard does not say whether write requests for more than {PIPE_BUF} bytes will be atomic, but requires that writes of {PIPE_BUF} or less 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, nbyte, 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}), the standard 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 best shown in 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.

The Working Group decided not to make an exception regarding partial writes when O_NONBLOCK is set. With the exception of writing to an empty pipe, the standard 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

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]

application should be prepared to handle partial writes on other kinds of file descriptors.

Where the standard requires −1 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 the standard). The error indications in the standard 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 Working Group chose to make 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 discussed by the Working Group. The Group decided this concept would unnecessarily limit application writing.

See also the discussion of O_NONBLOCK in **Input and Output Primitives** §B.6.

The standard 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.

The standard does not specify behavior of concurrent writes to a file from multiple processes. Applications should use some form of concurrency control.

References to actions taken on an "unrecoverable error" have been removed. It is considered beyond the scope of this standard to describe what happens in the case of hardware errors.

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*() §3.1.1 or an F_FDUPFD operation with *fcntl*() §6.5.2). 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 the standard, 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 the standard. See also **O_NONBLOCK** §B.6.

B.6.5.2 File Control. The ellipsis in the Synopsis is the syntax specified by the C Standard for a variable number of arguments. It is used because System V uses pointers for the implementation of file locking functions.

POSIX permits concurrent read and write access to file data using the *fcntl*() function; this is a change from the /usr/group Standard and previous 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 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*() §3.1.1 or *exec* §3.1.2 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 the standard to use different specifications.

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 this standard. 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*. 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, 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 shareable 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 setgid bit in most implementations; this was confusing, at best.

(2) 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*() §3.4.1 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.

The *l_start* element of the *flock* structure and the *offset* argument of *lseek*() are, in some cases, taken as signed offsets from some position in a file, but the type of these objects is allowed to be unsigned. This apparent conflict is avoided by the C Standard's definitions of conversions from signed to unsigned and of arithmetic operations on unsigned types. If U is of type *off_t*, the expressions

```
U + ((off_t) (-i))
```

and

```
U - i
```

will produce the same result, and, for example,

```
lseek (fd, (off_t) - 4, SEEK_END);
```

is well defined.

B.6.5.3 Reposition Read/Write File Offset. The C Standard includes the functions *fgetpos*() §B.6.5.3 and *fsetpos*() §B.6.5.3 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 that may extend the size of the file is *write*() §6.4.2, several Standard C functions, such as *fwrite*(), *fprintf*(), etc., may do so (by causing calls on *write*()).

An illegal file offset that would cause [EINVAL] to be returned may be both implementation-defined and device-dependent (for example, memory may have few illegal values). A negative file offset may be legal for some devices in some implementations.

See *fcntl*() §B.6.5.2 for a explanation of the use of signed and unsigned offsets with *lseek*().

B.7 Device- and Class-Specific Functions. This section has probably undergone more debate and revision than any other in the standard. Numerous historical implementations were investigated, and at least four major proposals were made.

There are several sources of the difficulties 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 Berkeley systems added to the basic Version 7 *ioctl*() command set; some of these were for features such as job control that POSIX 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, but the Working Group did not want to supply unnecessary impediments to internationalization.

The *1984 /usr/group Standard* 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 the standard, 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.

Recognizing the pitfalls of the high level approach, the Working Group focused on 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 AT&T System III *termio* was agreed upon. The format of that structure, referred to as the *termios* structure, has formed the basis for the current section.

A method is needed to communicate with the system about the *termios* information. Proposals have included:

(1) The *ioctl*() function as in System V. This has the same problems as mentioned above for the Version 7 *ioctl*() function, and is basically identical to it. Another problem is that the direction of the command (whether information is written from or read into the third argument) is not specified: in historical implementations only the device driver knows for sure. This is a problem for networked implementations. It is also a problem that there is no size parameter to specify the variable size of the third argument, and similarly for its type.

(2) An *iocntl*() function with additional arguments specifying direction, type, and size. But these new arguments would not help application writers, who would have no control over their values, which would have to match each command exactly. The new arguments do, however, solve the problems of networked implementations. And *iocntl*() is implementable in terms of *ioctl*() on historical implementations (without need for modifying existing code), although it is 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 would be 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 sidestepped for some time after the Trial Use Standard while the Working Group concentrated on the details of the information to be handled. The resulting specification resembles the information in System V, but attempts to avoid problems of case, speed, networks, and internationalization.

(5) Specific *tc**() functions to replace each *ioctl*() function were finally incorporated into the standard, instead of any of the above-mentioned proposals.

The issue of modem control was excluded from POSIX on the grounds that:
(1) It was concerned with setting and control of hardware timers;
(2) The appropriate timers and settings vary widely internationally;
(3) Feedback from X/Open indicated that this facility was not consistent with European needs, and that specification of such a facility was not a requirement for portability from their "international perspective."

B.7.1 General Terminal Interface. Although the Working Group attempted to take into account needs of both interface implementors and application developers throughout the standard, more attention was paid to the needs of the latter in this section. 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 interrupt level.

The *tc**() functions were introduced as a way of avoiding the problems inherent in the traditional *ioctl*() §B.7.1 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, 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:
(1) It allows strict type checking.
(2) The direction of transfer of control data is explicit.
(3) Portable capabilities are clearly identified.
(4) The need for a general interface routine is avoided.
(5) Size of the argument is well-defined (there is only one type).

The disadvantages include:
(1) No historical implementation uses the new method.
(2) There are many small routines instead of one general-purpose one.
(3) The historical parallel with *fcntl*() §6.5.2 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 **Control Modes** §7.1.2.4.

B.7.1.1.2 Process Groups.

B.7.1.1.3 The Controlling Terminal. The standard 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 this standard.

Traditional implementations allocate controlling terminals on certain *open*() calls. Since *open*() is part of the standard, its behavior had to be dealt with. The Working Group did not wish to require the traditional behavior, 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*() §B.5.3.1.

B.7.1.1.4 Terminal Access Control. The access controls described in this section 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 which 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 which 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** §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.

See also **job control** §B.2.3.

B.7.1.1.5 Input Processing and Reading Data.

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 non-blank 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 the standard 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.

B.7.1.1.7 Non-Canonical Mode Input Processing. Some points to note about MIN and TIME:

(1) In the preceding explanations one may notice that 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; while 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; while 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.

B.7.1.1.8 Writing Data and Output Processing.

B.7.1.1.9 Special Characters.

B.7.1.1.10 Modem Disconnect.

B.7.1.1.11 Closing a Terminal Device File.

B.7.1.2 Settable Parameters.

B.7.1.2.1 *termios* **Structure.** This structure is part of an interface which 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 the standard 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 ensure 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 worse in an escape sequence affecting the current character set, are beyond the scope of this standard and are best dealt with by the application processing the multibyte characters.

B.7.1.2.3 Output Modes. This standard 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 the standard 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.

B.7.1.2.5 Local Modes. Non-canonical mode is provided to allow fast bursts of input to be read efficiently while still allowing single character input.

The ECHONL function has historically been in many implementations. Since there seems to be no technical problem with supporting ECHONL it is included in the standard 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 existing implementations. Only when backwards compatibility of object code is a serious concern to an implementor should an implementation continue this practice. Correct applications which 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 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. Secondly, and 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.

The standard specifies that if an attempt to set the input baud rate to zero is made by *cfsetispeed*(), the input baud rate will be instead set to the output baud rate by *cfsetispeed*(). This allows implementations to provide support for split baud rates or not.

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 bad idea 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 section on access from processes in background process groups controls apply only to a process that is accessing its controlling terminal. (See **Terminal Access Control** §B.7.1.1.5).

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 which 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. Not doing this is at best antisocial. See also **General Terminal Interface** §B.7.1.

B.7.2.1 Get and Set State.

B.7.2.2 Line Control Functions.

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.

B.7.2.4 Set Foreground Process Group ID. The *tcsetpgrp*( ) function has identical functionality to the 4.2BSD *ioctl*( ) function TIOCSPGRP except for the additional security restrictions that the referenced terminal must be the controlling terminal for the calling process and the specified new process group must be currently in use in the caller's session.

See the discussion of C functions in **POSIX and the C Standard** §B.1.4.

B.8 Language-Specific Services for the C Programming Language. Common usage may be defined by historical publications as *The C Programming Language*, by Kernighan and Ritchie, listed in **Bibliographic Notes** §B.11.

The null set of supported languages is allowed.

The list of functions comprises the list of "common usage" functions, plus those that are not in common usage that are addressed by this standard. The rules for common usage conformance to the standard address whether the functions that are not generally considered in common usage are implemented. There are a large number of functions found in various systems that, although frequently found, are not broadly enough available to be considered in common usage. The *signal*( ) function (although in common usage) is omitted because it is the belief of the working group that programs conforming to POSIX should use *sigaction*( ) instead.

B.8.1 Referenced C Language Routines.

B.8.1.1 Extensions to Time Functions. System V uses the **TZ** environment variable to set some information about time. It has the form (spaces inserted for clarity):

std offset dst

where the first three characters (*std*) are the name of the standard time zone, the digits which follow (*offset*) represent the time added to the local time zone to arrive at Coordinated Universal Time, and the next three characters (*dst*) are the name of the summer time zone. The meaning of *offset* implies that most sites west of the Prime Meridian will have a positive offset (preceded by an optional plus sign, "+"), while most sites east of the Prime Meridian will have a negative offset (preceded by a minus sign, "–"). Both *std* and *offset* are required; if *dst* is missing, summer time does not apply.

Currently, the UNIX system *localtime*( ) function translates a number of **seconds since the Epoch** §2.3 into a detailed breakdown of that time. This breakdown includes:

(1) Time of day: Hours, minutes, and seconds.
(2) Day of the month, month of the year, and the year.
(3) Day of the week and day of the year (Julian day).
(4) Whether or not summer (daylight saving) time is in effect.

It is the first and last items that present a nasty problem: The time of the day depends on whether or not summer time is in effect. Whether or not summer time is in effect depends on the locale and date.

Most historical systems had time zone rules compiled into the C library. These rules usually represented United States rules for 1970 to 1986. This did not accommodate the changes of 1987, nor other world variations (½-hour time, double daylight time, and solar time being common but not complete examples). Some recent systems addressed these problems in various ways.

Having the rules compiled into the program made binary distributions that accommodated all the variations (including sudden changes to the law), and per-process rule changes, difficult at best.

The current standard includes a way of specifying the time zone in the **TZ** string, but only permits one time zone pattern at a time, thus not dealing with different patterns in previous years, and does not deal with such issues as solar time. Methods exist to deal with all the problems above. The method in this standard appears to be simpler to implement and may be faster in execution when it is adequate.

The current standard also permits an implementation-defined rule set which begins with a colon. (The previous format cannot begin with a colon.)

Rules of the form AAAn or AAAnBBB (the style used in many historical implementations) do not carry with them any statement about the start and end of daylight time (neither the date nor the time of day; the default to 02:00 not applying if no *rule* is present at all), thus implying that the implementation must provide the appropriate *rule*s. An implementation may provide those rules in any way it sees fit, as long as the constraints implied by the **TZ** string as provided by the user are met. Specifically the implementation may use the string as an index into a table, which may reside either on disk or in memory. Such tables could contain rules which are sensitive to the year to which they are applied, again since the user did not specify the exact *rule*. (Although impractical, every possible **TZ** string could be represented in a table, as a detail of implementation; the less specific the user is about the **TZ** string, the more freedom the implementation has to interpret it.)

There is at least one public domain time zone implementation (the Olson/Harris method) that uses non-specific **TZ** strings and a table, as described above and handles all the general time zone problems mentioned above. This implementation also appears in a late release of 4.3BSD. If this implementation honors all the specifications provided in the **TZ** string it would conform to the standard. Nothing precludes the implementation from adding information beyond that given by the user in the **TZ** string.

The fully-specified **TZ** environment variable extends the historical meaning to also include a rule for when to use standard time and when to use summer time. Southern hemisphere time zones are supported by allowing the first *rule date* (change to summer time) to be later in the year than the second *rule date* (change to standard time).

This mechanism accommodates the "floating day" rules (for example "last Sunday in October") used in the U.S. and Canada (and the European Economic Community for the last several years). In theory, **TZ** only has to be set once and

then never touched again unless the law is changed.

Julian dates are proposed with two syntaxes, one zero-based, the other one-based. They are here for historical reasons. The one-based counting (*J*) is used more commonly in Europe (and on calendars people may use for reference). The zero-based counting (*n*) is used currently in some implementations and should be kept for historical reasons as well as being the only way to specify Leap day.

It is expected the leading colon format will allow systems to implement an even broader range of specifications for the time zone without having to resort to a file, or to permit naming an explicit file containing the appropriate rules.

The specification in the standard for **TZ** assumes that very few programs need to be historically accurate as long as the relative timing of two events is preserved.

Summer time is governed by both locale and date. This proposal only handles the locale dependency. Using an implementation-defined file format for either the entire **TZ** variable or to specify the *rules* for a particular time zone is allowed as a means by which both the locale and date dependency can be handled.

Since historical implementations do not examine **TZ** beyond the assumed end of *dst*, it is possible to literally extend **TZ** and break very little existing software. Since much historical software doesn't work anyway outside the U.S. time zones, minor changes to **TZ** (such as extending *offset* to be *hh*:*mm*—as long as the colon and minutes, *:mm*, are optional) should have little effect.

B.8.1.2 Extensions to *setlocale*() Function. The C Standard defines a collection of interfaces to support internationalization. One of the most significant aspects of these interfaces is a facility to set and query the *international environment*. The international environment is a repository of information that affects the behavior of certain functionality, namely:

(1) Character Handling
(2) String Handling (i.e., collating)
(3) Date/Time Formatting
(4) Numeric Editing.

The *setlocale*() function provides the application developer with the ability to set all or portions, called *categories*, of the international environment. These categories correspond to the areas of functionality, mentioned above. The syntax for *setlocale*() is the following:

```
char   *setlocale (category, locale)
int    category;
char   *locale
```

where *category* is the name of one of five categories, namely:

LC_CTYPE
LC_COLLATE
LC_TIME
LC_MONETARY
LC_NUMERIC

In addition, a special value, called LC_ALL, directs *setlocale*() to set all categories.

The *locale* argument is a character string that points to a specific setting for the international environment, or locale. There are three preset values for the locale argument, namely:

"C"	Specifies the minimal environment for C translation. If *setlocale*() is not invoked, the "C" locale is the default.
""	Specifies an implementation-defined native environment.
NULL	Used to direct *setlocale*() to query the current international environment and return the name of the locale.

This section describes the behavior of an implementation of *setlocale*() and its use of environment variables in controlling this behavior on POSIX-based systems. There are two primary uses of *setlocale*():

(1) Querying the international environment to find out what it is set to;

(2) Setting the international environment, or *locale*, to a specific value.

The following subsections describe the behavior of *setlocale*() in these two areas. Since it is difficult to describe the behavior in words, examples will be used to illustrate the behavior of specific uses.

To query the international environment, *setlocale*() is invoked with a specific category and the **NULL** pointer as the locale. The **NULL** pointer is a special directive to *setlocale*() that tells it to query rather than set the international environment. Below is the syntax for using *setlocale*() to query the name of the international environment:

$$setlocale(\left\{\begin{array}{l} LC\_ALL \\ LC\_CTYPE \\ LC\_COLLATE \\ LC\_TIME \\ LC\_NUMERIC \\ LC\_MONETARY \end{array}\right\}, (char\ *)\ NULL);$$

The *setlocale*() function returns the string corresponding to the current international environment. This value may be used by a subsequent call to *setlocale*() to reset the international environment to this value. However, it should be noted that the return value from *setlocale*() is a pointer to a static area within the function and is not guaranteed to remain unchanged (i.e., it may be modified by a subsequent call to *setlocale*()). Therefore, if the purpose of calling *setlocale*() is to save the value of the current international environment so it can be changed and reset back later, the return value should be copied to a character array in the calling program.

There are three ways to set the international environment with *setlocale*():

`setlocale(`*category*`,` *string*`)`

This usage will set a specific *category* in the international environment to a specific value corresponding to the value of the *string*. A specific example is provided below:

```
setlocale(LC_ALL, "Fr_FR.8859");
```

In this example, all categories of the international environment will be set to the locale corresponding to the string "Fr_FR.8859", or the French language as spoken in France using the ISO 8859/1 code set.

If the string does not correspond to a valid locale, *setlocale*() will return a **NULL** pointer and the international environment is not changed. Otherwise, *setlocale*() will return the name of the locale just set.

`setlocale(`*category*`, "C")`
The C Standard draft proposal states that one locale must exist on all conforming implementations. The name of the locale is "C", and corresponds to a minimal international environment needed to support the C programming language.

`setlocale(`*category*`, "")`
This will set a specific category to an implementation-defined default. For POSIX-based systems, this corresponds to the value of the environment variables.

B.8.2 FILE-Type C Language Functions.

B.8.2.1 Map a Stream Pointer to a File Descriptor. Without some specification of which file descriptors are associated with these streams, it is impossible for an application to set up the streams for another application it starts with *fork*() §3.1.1 and *exec* §3.1.2. In particular, it would not be possible to write a portable version of the `sh` command processor (although there may be other constraints that would prevent that portability).

B.8.2.2 Open a Stream on a File Descriptor. The file descriptor may have been obtained from *open*() §5.3.1, *creat*() §5.3.2, *pipe*() §6.1.1, *dup*() §6.2.1, *fcntl*() §6.5.2, or inherited through *fork*() §3.1.1 or *exec* §3.1.2, or perhaps obtained by implementation-dependent means, such as the 4.3BSD *socket*() call.

The meanings of the *type* arguments of *fdopen*() and *fopen*() differ. With *fdopen*(), open for write ("w" or "w+") does not truncate and append ("a" or "a+") cannot create for writing. There is no need for "b" in the format due to the equivalence of binary and text files in POSIX. See **Text vs. binary file modes** §B.1.4.

B.8.2.3 Interactions of Other FILE-Type C Functions. Note that the existence of open streams on a file implies open file descriptors, and thus affects the timestamps of the file. The intent is that using *stdio* routines to read a file must eventually update the access time, and using them to write a file must eventually update the modify and change times. However, the exact timing of marking the *st_atime*, *st_ctime*, and *st_mtime* fields cannot be specified, as that would be tantamount to mandating a particular buffering strategy.

The purpose of the rules about handles is to give the application writer a fighting chance of writing a program that uses *stdio* and does some shell-like things, in particular create an open file for a child process to use, where both the parent and child wish to use *stdio*, with the consequences of buffering. This in most cases cannot happen in Standard C (because there is no way to create a second handle), but the *system*() function can cause this to occur, at least in

most historical implementations.

There are some implied rules about inter-process synchronization, but no mechanism is given, intentionally. In the simplest case, if the parent meets the requirements on all its files, and then performs a *fork*() and a *wait*() before further activity on them (and a *fflush*() on input files after that), the synchronization desired will be achieved. Synchronization could in theory be done with signals, but the only likely case is the one just described. The terms *handle* and *active handle* were required to make the text readable, and are not intended for use outside this discussion.

Note that since *exit*() implies *_exit*(), a file descriptor is also closed by *exit*().

Because a handle is either freshly opened, or if not must have handed off control of the open file description as specified, the new handle is always ready to be used (except for seeks), with no initialization. (A freshly opened stream has not yet done any reads, as required by Standard C, at least implicitly by the rules associated with *setvbuf*().)

In requiring the seek to an appropriate location for the new handle, the application is required to know what it is doing if it is passing streams around with seeks involved. If the required seek is not done, the results are undefined (and in fact the program probably won't work on many common implementations).

A naive program used as a utility can be reasonably expected to work properly when the constraints are met by the calling program, because it will not hand off file descriptors except with closes.

The *exec* functions are treated specially, because the application should always *fflush*() everything before performing one of the *exec* functions. If *stdout* is available on the same open file description after the *exec*, it is a different stream, at least because any unflushed data will be discarded during the *exec*. (Similarly for *stdin*.) Process termination is also special because a process terminating due to a signal or *_exit*() will not have the buffers flushed.

The standard does not specify asynchronous I/O, and when dealing with asynchronous I/O the problem of coordinating access to streams will be more difficult. If asynchronous I/O is provided as an extension, the problems it introduces in this area should be addressed as part of that extension.

It may be that functions such as *system*() and *popen*(), currently being considered by P1003.2, will have to perform some of these operations.

The introduction of underlying functions allows generic reference to *errno* values returned by those functions, and also to other side effects (as required in the *handles* discussion above). It is not intended to specify implementation, although many implementations may in fact use those functions. Standard C says very little about *errno* in the context of *stdio*. In the more restricted POSIX environment, providing a reasonable set of *errno* values become possible.

Specifying the semantics with respect to line buffered streams is difficult because specifying exactly what a line is gets complex. Rather than respecify it, the semantics of *fgets*() and *fputs*() are used, as they have the right conceptual effect. The problem with this is that *ordinary files* require some kind of readahead to find the separating linefeed. Thus the addition that any implied buffering doesn't exist for the purposes of the discussion. In a typical system, the real meaning is "seek back to the last linefeed you actually consumed, assuming you

read the file beyond it in the first place." This cannot be said directly because it presumes an ordinary file, which is not presumed in this section; remote access to a line-oriented file might be likely. Line-buffered output streams don't have the problem because they cannot buffer back, by definition.

B.8.2.3.1 *fopen*().

B.8.2.3.2 *fclose*(). The *fclose*() function is required to synchronize the buffer pointer with the file pointer (unless it already is, which would be the case at EOF). Functionality equivalent to:

```
fseek(stream, ftell(stream), SEEK_SET)
```

does this nicely. The exception for devices incapable of seeking is an obvious requirement, but the implication is that there is no way to reliably read a buffered pipe and hand off handles. This is reality as it is in historical implementations, and is inherent in any "readahead" buffering scheme. This limitation is also reflected in the handle hand-off rules.

Note that the last byte read from a stream, and the last byte read from an open file description are not necessarily the same; in most cases the open file description's pointer will be past that of the stream because of the stream's readahead.

B.8.2.3.3 *freopen*().

B.8.2.3.4 *fflush*(). The *fflush*() function is required to flush its buffers on input, and resynchronize the buffer pointer. The reasoning is quite similar to *fsync*(). Standard C already requires this for input.

B.8.2.3.5 *fgetc*(), *fgets*(), *fread*(), *getc*(), *getchar*(), *gets*(), *scanf*(), *fscanf*().

B.8.2.3.6 *fputc*(), *fputs*(), *fwrite*(), *putc*(), *putchar*(), *puts*(), *printf*(), *vprintf*(), *vfprintf*().

B.8.2.3.7 *fseek*(), *rewind*(). The *fseek*() function must operate as specified, to make the case where seeking is being done work. The key requirement is to avoid an optimization such that an *fseek*() would not result in an *lseek*() if the *fseek*() pointed within the current buffer. This optimization is valuable in general, so it is only required after an *fflush*().

B.8.2.3.8 *perror*().

B.8.2.3.9 *tmpfile*().

B.8.2.3.10 *ftell*().

B.8.2.3.11 Error Reporting. [ENOMEM] was considered for addition as an explicit possible error, because most implementations use *malloc*(). This was not done because the scope does not include "out of resource" errors. Nevertheless this is the most likely error to be added to the possible error conditions. Other implementation-defined errors, particularly in the *f*open*() family, are to be expected, and the generic rules about adding (or deleting) possible errors apply, except that it is expected that implementation-defined changes in the error set returned by *open*() would also apply to *fopen*() (unless the condition can't possibly happen in *fopen*(), which may be possible, but appears unlikely.).

B.8.2.3.12 *exit*(), *abort*().

B.8.2.4 Operations on Files — the *remove*() Function.

B.8.3 Other C Language Functions.

B.8.3.1 Non-Local Jumps. The C Standard specifies various restrictions on the usage of the *setjmp*() macro in order to permit implementors to recognize the name in the compiler and not implement an actual function. These same restrictions apply to the *sigsetjmp*() macro.

There are processors that cannot easily support these calls, but the Working Group did not consider that a sufficient reason not to include them.

The distinction between *setjmp*()/*longjmp*() and *sigsetjmp*()/*siglongjmp*() is only significant for programs which use the *sigaction*(), *sigprocmask*(), or *sigsuspend*() functions. Since earlier implementations did not have signal masks, only a single pair was provided.

4.2BSD and 4.3BSD systems provide functions named *_setjmp*() and *_longjmp*() which, together with *setjmp*()/*longjmp*(), provide the same functionality as *sigsetjmp*()/*siglongjmp*(). On those systems, *setjmp*()/*longjmp*() save and restore signal masks, while *_setjmp*()/*_longjmp*() do not. On System V Release 3 and in corresponding issues of the *SVID*, *setjmp*()/*longjmp*() are explicitly defined not to save and restore signal masks. In order to permit existing practice in both cases, the Working Group decided not to specify the relation of *setjmp*()/*longjmp*() to signal masks and to define a new set of functions instead.

B.8.3.2 Set Time Zone.

B.9 System Databases. At one time, this chapter was entitled **Passwords**, but this title was changed as all references to a "password file" were changed to refer to a "user database."

B.9.1 System Databases. There are no references in the standard to a *passwd file* or a *group file* and there is no requirement that the *group* or *passwd* databases be kept in ASCII files. Many large timesharing systems use *passwd* databases that are hashed for speed. Certain security classifications prohibit certain information in the *passwd* database from being publicly readable.

The encoded password fields were deleted from both the *passwd* and *group* databases in order to meet the requirements of the US Government NBS Password FIPS (Publication 112, *Password Usage*, dated May 30, 1985, and FIPS concerns in general).

The term "encoded" is used instead of "encrypted" in order to avoid the implementation connotations (such as reversibility, or use of a particular algorithm) of the latter term.

The functions *getgrent*(), *setgrent*(), *endgrent*(), *getpwent*(), *setpwent*(), and *endpwent*() are not included in this standard because they provide a linear database search capability that is not generally useful (the *getpwuid*(), *getpwnam*(), *getgrgid*(), and *getgrnam*() functions are provided for keyed lookup), and because in certain distributed systems, especially those with different authentication domains, it may not be possible or desirable to provide an application with the ability to browse the system databases indiscriminately.

B.9.2 Database Access.
B.9.2.1 Group Database Access.
B.9.2.2 User Database Access.

B.10 Data Interchange Format.

B.10.1 Archive/Interchange File Format. There are three areas of interest associated with file interchange:

(1) **Media.** There are other existing standards that define the media used for data interchange.

(2) **User Interface.** This rightfully should be in the IEEE Std 1003.2 standard.

(3) **Format of the Data.** None of the 1003 Working Groups address topics that match this area. The Working Groups felt that this area is closest to the types of things that should be in the IEEE Std 1003.1-1988 document, as the level of that document most closely matches the level of data required.

There appear to be two programs in wide use today, `tar` and `cpio`. There are large camps of supporters for each program. Four options were considered for the standard:

(1) Make both formats optional. This was considered unacceptable because it does not allow any portable method for data interchange.

(2) Require one format.

(3) Require one format with the other optional.

(4) Require both formats.

Both the Extended `cpio` and the Extended `tar` Formats are required by this standard.

There are a number of concerns about defining extensions that are known to be required by existing implementations. Failure to specify a consistent method to implement these extensions will severely limit portability of the program and, more importantly, will create severe confusion if these extensions are later standardized.

Two of these extensions that the Working Group felt should be documented are symbolic links, that were defined by 4.2BSD and 4.3BSD systems, and high performance (or contiguous) files, that exist in a number of implementations and are now being considered for the 1003.4 standard.

By defining these extensions, implementors are able to recognize these features and take appropriate implementation-defined actions for these files. For example, a high performance file could be converted to a regular file if the system didn't support high performance files; symbolic links might be replaced by normal hard links.

The Working Group has held to the policy of not defining user interfaces to utilities by avoiding any description of a `tar` or `cpio` command. The behavior of the former command was described in some detail in previous drafts.

The possibilities for transportable media include, but are not limited to:

(1) ½-inch magnetic tape, 9 track, 1600 BPI

(2) ½-inch magnetic tape, 9 track, 6250 BPI

(3) QIC-11, ¼-inch streamer tape

(4) QIC-24, ¼-inch streamer tape

(5) 5.25-inch floppies, 9 512-byte sectors/track, 96 TPI
(6) 5.25-inch floppies, 9 512-byte sectors/track, 48 TPI
(7) IBM 3480 cartridges.

Specification of such media was considered part of the scope of the Trial Use Standard, but has been excluded from the Full Use Standard.

The utilities are not restricted to work only with *transportable* media: existing related utilities are often used to transport data from one place to another in the file hierarchy.

The formats are included to provide implementation-independent ways to move files from one system to another and also to provide ways for a user to save data on a transportable medium to be restored at a later date. Unfortunately, these two goals can contradict each other as system security problems are easy to find in tape systems if they are not protected. Thus there are strict requirements about how the mechanism to copy files shall react when operated by both privileged and nonprivileged users. The general concept is that a privileged (historically using the ISUID bit in the file's mode with the owner UID of the file set to super-user) version of the utility (or one operated by a privileged user) can be used as a save/restore scheme, but a nonprivileged version is used to interpret media from a different system without compromising system security.

Regardless of the archive format used, guidelines should be observed when writing tapes to be read on other systems. Assuming the target system is POSIX comformant, archives created should only use definitions found in POSIX (e.g., file types, minimum values as found in Chapter 2) and should only use relative pathnames (i.e., no leading slash).

Both `tar` and `cpio` formats have traditionally been used for both exchange of information and archiving. These formats have a number of features that facilitate archiving, for example, the ability to store information about a file which is a device. This standard does not assume this kind of data is portable. It is intended that these formats provide for the portable exchange of source information between dissimilar systems. This requires specification of the character set to be used (ASCII) when these formats are used to write source information.

All data written by format-creating utilities and read by format-reading utilities is an ordered stream of bytes. The first byte of the stream should be first on the medium, the second byte second, etc. On systems where the hardware swaps bytes or otherwise rearranges the byte stream on output or input, the implementor of these utilities must compensate for this, so that the data on the storage device retains its ordered nature.

This standard describes two different formats for data archiving and interchange. Through the balloting process the Working Group found strong support for both formats. This is a clear indication of the need for both formats due to existing practice. The balloting process has also defined a number of deficiencies of each format. The strong support indicates that these deficiencies are not sufficient to remove either format from the standard, but will need to be addressed by the Working Group in the future. It was not practical to remedy these deficiencies during the balloting process. Considerable thought and review must occur before making any changes to these formats. It was felt that

the best solution is to advise implementors and application writers of these deficiencies by documenting them in the rationale and to include both formats in the standard.

The Working Group recognizes the desirability for migration toward one common format and has been made aware of some strong inputs to consider both formats in light of existing practice, current technology trends and the 1003 standards activities such as security and high performance systems to develop one format that is technically superior. This format will be incorporated into a future version of this standard when it is developed.

The deficiencies that have been identified in the existing formats are as follows. The size of a file link is limited to 100 characters in `tar`. A number of fields in the `cpio` header (*c_filesize*, *c_dev*, *c_ino*, *c_mode*, and *c_rdev*) are too short to support values that the standard allows these fields to contain. Some existing implementations and current trends in development will require the ability to represent even larger values in these fields. The `cpio` format does not provide a mechanism to represent the user and group IDs symbolically, and a range of implementation-defined file types have not been reserved for the user. The `cpio` format specification does not reserve any formats for implementation-defined usage. The extensions that have been made to `cpio` for this standard are compatible with existing versions of `cpio`. Correction of some of these deficiencies would make existing versions of `cpio` behave unpredictably. When these changes are made the `cpio` magic number will have to be changed.

This chapter uses the term *file name*, which is actually a defined term in P1003.2. Note that *filename* and *file name* are not synonyms; the latter is a synonym for *pathname*, in that it includes the slashes between filenames.

In earlier drafts, the word "local" was used in the context of "file system" and was taken (incorrectly) to be related to "remotely-mounted file system." This was not intended. The term "(local) file system" refers to the file hierarchy as seen by the utilities, and "local" was removed because of this confusion.

B.10.1.1 Extended `tar` Format. The original model for this facility is the 4.3BSD or Version 7 `tar` program and format, but the format given here is an extension of the traditional `tar` format. The name USTAR was adopted to reflect this.

This description reflects numerous enhancements over previous versions. The goal of these changes was not only to provide the functional enhancements desired, but to retain compatibility between new and old versions. This compatibility has been retained. Archives written using the old archive format are compatible with the new format. Archives written using this new format may be read by applications designed to use the old format as long as the functional enhancements provided here are not used. This means the user is limited to archiving only regular type files and nonsymbolic links to such files.

Implementors should be aware that the previous file format did not include a mechanism to archive directory type files. For this reason, the convention of using a file name ending with slash was adopted to specify a directory on the archive.

Note that the total size of the name and prefix fields have been set to meet the minimum requirements for {PATH_MAX}. If a pathname will fit within the name field, it is recommended that the pathname be stored there without the use of the prefix field. Although the name field is known to be too small to contain {PATH_MAX} characters, the value was not changed in this version of the archive file format to retain backward compatibility and instead the *prefix* was introduced. Also because of the earlier version of the format, there is no way to remove the limitation on the *linkname* field being limited in size to just that of the name field.

The *size* field is required to be meaningful in all implementation extensions, although it could be zero. This is required so that the data blocks can always be properly counted.

It is suggested that if device special files need to be represented which cannot be represented in the standard format that one of the extension types ('A'–'Z') be used, and that the additional information for the special file be represented as data, and be reflected in the size field.

Attempting to restore a special file type, where it is converted to ordinary data and it conflicts with an existing file name, need not be specially detected by the utility. If run as an ordinary user, a format-reading utility should not be able to overwrite the entries in (say) `/dev` in any case (whether the file is converted to another type or not). If run as a privileged user, it should be able to do so, and it would be considered a bug if it did not. The same is true of ordinary data files and similarly-named special files; it is impossible to anticipate the user's needs (who could really intend to overwrite the file), so the behavior should be predictable (and thus regular) and rely on the protection system as required.

The values '2' and '7' in the typeflag field are intended to define how symbolic links and contiguous files can be stored in a `tar` archive. The standard does not require the symbolic link or contiguous file extensions, but does define a standard way of archiving these files so that all conforming systems can interpret these file types in a meaningful and consistent manner. On a system which does not support extended file types, the format-interpreting utility should do the best it can with the file and go on to the next.

B.10.1.2 Extended `cpio` Format. The model for this format is the existing System V `cpio -c` data interchange format. This model documents the portable version of `cpio` format and not the binary version. It has the flexibility to transfer data of any type described within the POSIX standard, yet is extensible to transfer data types specific to extensions beyond POSIX (e.g., symbolic links or contiguous files). Because it describes existing practice, there is no question of maintaining upward compatibility.

This section does not standardize behavior for the utility when the file type is not understood or supported. It is useful for the utility to report to the user whatever action is taken in this case, though the standard neither requires nor recommends this.

B.10.1.2.1 Header. There has been some concern that the size of the *c_ino* field of the header is too small to handle those systems which have very large i-node numbers. However, the *c_ino* field in the header is used strictly as a hard link resolution mechanism for archives. It is not necessarily the same value as the i-node number of the file in the location that file is extracted from.

B.10.1.2.2 File Name. For most current implementations of the `cpio` utility, {PATH_MAX} bytes can be used to describe the pathname without the addition of any other header fields (the null byte would be included in this count). {PATH_MAX} is the minimum value for pathname size, documented as 256 bytes in Chapter 2 of the standard. However, an implementation may use *c_namesize* to determine the exact length of the pathname. With the current description of the `cpio` header, this pathname size can be as large as a number which is described in six octal digits.

B.10.1.2.3 File Data.

B.10.1.2.4 Special Entries. These are provided to maintain backward compatibility.

B.10.1.2.5 `cpio` Values. Three values are documented under the *c_mode* field values to provide for extensibility for known file types:

0110000 Reserved for contiguous files. The implementation may treat the rest of the information for this archive like a regular file. If this file type is undefined, the implementation may create the file as a regular file.

0120000 Reserved for files with symbolic links. The implementation may store the link name within the data portion of the file. If this type is undefined, the implementation may not know how to link this file or be able to understand the data section. The implementation may decide to ignore this file type and output a warning message.

0140000 Reserved for sockets. If this type is undefined on the target system, the implementation may decide to ignore this file type and output a warning message.

This provides for extensibility of the `cpio` format while allowing for the ability to read old archives. Files of an unknown type may be read as "regular files" on some implementations. On a system which does not support extended file types, the format-interpreting utility should do the best it can with the file and go on to the next.

B.10.1.3 Multiple Volumes. Multi-volume archives have been introduced in a manner that has become a *de facto* standard in many implementations. Though it is not required by POSIX, classical implementations of the format-reading and -creating utility, upon reading logical end-of-file, check to see if an error channel is open to a controlling terminal. The utility then produces a message requesting a new medium to be made available. The utility waits for a new medium to be made available by attempting to read a message to restart from the controlling terminal. In all cases, the communication with the controlling terminal is in an implementation-defined manner.

The section **Multiple Volumes** §10.1.3 is intended to handle the issue of multiple volume archives. Since the end-of-medium and transition between media

are not properly part of this standard, the transition is described in terms of files; the word "file" is used in a very broad, but correct, sense—a tape drive is a file.

The intent is that files will be read serially until the end-of-archive indication is encountered, and that file or media change will be handled by the utilities in an implementation-defined manner.

Note that there was an issue with the representation of this on magnetic tape, and the standard is intended to be interpreted such that each byte of the format is represented on the media exactly once. In some current implementations, it is not deterministic whether encountering the end-of-medium reflector foil on magnetic tape during a write will yield an error during a subsequent *read*() of that record, or if that record is actually recorded on the tape. It is also possible that *read*() will encounter the end-of-medium when end-of-medium was not encountered when the data was written. This has to do with conditions where the end of [magnetic] record is in such a position that the reflector foil is on the verge of being detected by the sensor and is detected during one operation and not on a later one, or vice-versa.

An implementation of the format-creating utility must assure when it writes a record that the data appears on the tape exactly once. This implies that the program and the tape driver work in concert. An implementation of the format-reading utility must assure that an error in a boundary condition described above will not cause loss of data.

The general consensus was that the following would be considered as correct operation of a tape driver when end-of-medium is detected:

(1) During writing, either:

(a) The record where the reflector spot was detected is backspaced over by the driver so that the trailing tape mark that will be written on *close*() will overwrite. Writing the tape mark should not yield an end-of-medium condition.

(b) Or, the condition is reported as an error on the *write*() following the one where the end-of-medium is detected (the one where the end-of-medium is actually detected completing successfully). No data will be actually transferred on the *write*() reporting the error. The subsequent *close*() would *write*() a tape mark following the last record actually written. Writing the tape mark, and writing any subsequent records, should not yield any end-of-medium conditions.

(The latter behavior permits the implementation of ANSI standard labels because several records (the trailer records) can be written after the end-of-medium indications. It also permits dealing with, for example, COBOL "ON" statements.)

(2) During reading, the end-of-medium indicator is simply ignored, presuming that a tape mark (end-of-file) will be recorded on the magnetic medium, and the reflector foil was advisory only to the *write*().

Systems where these conditions are not met by the tape driver should assure that the format-creating and -reading utilities assure proper representation and interpretations of the files on the media, in a way consistent with the above recommendations.

The typical failures on systems that do not meet the above conditions are either:

(1) To leave the record written when the end-of-medium is encountered on the tape, but to report that it was not written. The format-creating utility would then rewrite it, and then the format-reading utility could see the record twice if the end-of-medium is not sensed during the read operations.

(2) Or, the *write*() occurs uneventfully, but the *read*() senses the error and does not actually see the data, causing a record to be omitted.

Nothing in this standard requires that end-of-medium be determined by anything on the medium itself (for example, a predetermined maximum size would be an acceptable solution for the format creating utility). The format-reading utility must be able to *read*() tapes written by machines that do use the whole medium, however.

On media where end-of-medium and end-of-file are reliably coincident, such as disks, end-of-medium and end-of-file can be treated as synonyms.

Note that partial physical records (corresponding to a single *write*()) can be written on some media, but that only full physical records will actually be written to magnetic tape, given the way the tape operates.

B.11 Bibliographic Notes. There are far more related papers and books than are mentioned here, and some of them may be as good or better.

B.11.1 Related Standards. The standard assumes that any terms not defined in Chapter 2 are defined in the *IEEE Standard Dictionary of Electrical and Electronics Terms,* IEEE Std 100-1977.

The *1984 /usr/group Standard* may be ordered from

/usr/group Standards Committee
4655 Old Ironsides Drive, Suite 200
Santa Clara, California 95054
(408) 986-8840

The basic historical reference on the C language is

- Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey (1978).

B.11.2 Historical Implementations. A principal ancestor of all the historical implementations is the Multics System

- Organick, Elliot I., *The Multics System: An Examination of Its Structure,* The MIT Press, Cambridge, MA (1972).

The most basic and influential paper on historical implementations is

- Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System," *Bell System Technical Journal* **57**(6 Part 2) pp. 1905-1929 American Telephone and Telegraph Company, (July-August 1978). This is a revised version and describes Version 7.
- Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System," *Commun. ACM* **7**(7) pp. 365-375 Association for Computing Machinery, (July 1974). This is the original paper, which describes Version 6.

The Version 7 manual is

- AT&T, *UNIX Time Sharing System: UNIX Programmer's Manual, Seventh Edition,* Bell Telephone Laboratories, Inc., Murray Hill, New Jersey (January, 1979).

Dennis Ritchie has also done several papers on the history and evolution of the system

- Ritchie, Dennis, "The Evolution of the UNIX Time-sharing System," *AT&T Bell Laboratories Technical Journal* **63**(8) pp. 1577-1593 American Telephone and Telegraph Company, (October 1984).
- Ritchie, Dennis M., "Reflections on Software Research," *Commun. ACM* **27**(8) pp. 758-760 Association for Computing Machinery, (August 1984). ACM Turing Award Lecture.
- Ritchie, Dennis M., "Unix: A Dialectic," *Winter 1987 USENIX Association Conference Proceedings, Washington, DC*, pp. 29-34 USENIX Association, P.O. Box 2299, Berkeley, CA 94710, (21-23 January 1987).

Important collections of papers on the system may be found in

- BSTJ, "UNIX Time-Sharing System," *Bell System Technical Journal* **57**(6 Part 2)American Telephone and Telegraph Company, (July-August 1978).
- BLTJ, "The UNIX System," *AT&T Bell Laboratories Technical Journal* **63**(8 Part 2)American Telephone and Telegraph Company, (October 1984).

The System III manual is

- AT&T, *UNIX System III Programmer's Manual,* Western Electric Company, Inc., Greensboro, N.C. (October, 1981).

The *SVID*

- AT&T, *System V Interface Definition, Issue 2,* AT&T (1986).

may be ordered from:

AT&T Customer Information Center
Attn: Customer Service Representative
P.O. Box 19901
Indianapolis, IN 46219

(800) 432-6600 (Inside U.S.A.)
(800) 255-1242 (Inside Canada)
(317) 352-8557 (Outside U.S.A. and Canada)

using the following Select Codes:

320-011 Volume I
320-012 Volume II
320-013 Volume III
307-131 all three volumes

The implementation of System V is described in

- Bach, Maurice J., *The Design of the UNIX Operating System,* Prentice-Hall, Englewood Cliffs, New Jersey (1986).

The 4.3BSD manual

- UCB-CSRG, *4.3 Berkeley Software Distribution, Virtual VAX-11 Version,* The Regents of the University of California, Berkeley, California (April 1986).

is printed by the USENIX Association, and their members may order from them:

USENIX Association
P.O. Box 2299
Berkeley, CA 94710
(415) 528-8649

The implementation of the kernel of 4.3BSD is described in

- Quarterman, John S., Silberschatz, Abraham, and Peterson, James L., "4.2BSD and 4.3BSD as Examples of the UNIX System," *ACM Computing Surveys* **17**(4) pp. 379-418 Association for Computing Machinery, (December 1985).

- 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,* Addison-Wesley, Reading, Massachusetts (1988).

B.11.3 Historical Application Programming Tutorials. A useful tutorial on programming in the C language is

- Harbison, Samuel P. and Steele, Guy L., *C: A Reference Manual,* Prentice-Hall, Englewood Cliffs, New Jersey (1987).

A highly regarded book, though not one for beginners, is

- Kernighan, Brian W. and Pike, Rob, *The UNIX Programming Environment,* Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1984).

One more oriented towards Berkeley systems is

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

and a more recent one is

- Rochkind, Marc J., *Advanced UNIX Programming,* Prentice-Hall, Englewood Cliffs, New Jersey (1985).

Identifier Index

access() File Accessibility. {5.6.3} 100
alarm() Schedule Alarm. {3.4.1} 68
asctime() Extensions to Time Functions. {8.1.1} 142
cfgetispeed() Baud Rate Functions. {7.1.2.7} 134
cfgetospeed() Baud Rate Functions. {7.1.2.7} 134
cfsetispeed() Baud Rate Functions. {7.1.2.7} 134
cfsetospeed() Baud Rate Functions. {7.1.2.7} 134
chdir() Change Current Working Directory. {5.2.1} 85
chmod() Change File Modes. {5.6.4} 101
chown() Change Owner and Group of a File. {5.6.5} 102
close() Close a File. {6.3.1} 111
closedir() Directory Operations. {5.1.2} 83
cpio Extended `cpio` Format. {10.1.2} 159
creat() Create a New File or Rewrite an Existing One. {5.3.2} 90
ctermid() Generate Terminal Pathname. {4.7.1} 79
cuserid() Get User Name. {4.2.4} 74
directory Directory Operations. {5.1.2} 83
<dirent.h> Format of Directory Entries. {5.1.1} 83
dup() Duplicate an Open File Descriptor. {6.2.1} 110
dup2() Duplicate an Open File Descriptor. {6.2.1} 110
environ Execute a File. {3.1.2} 50
errno Error Numbers. {2.5} 37
<errno.h> Error Numbers. {2.5} 37
exec Execute a File. {3.1.2} 50
execl() Execute a File. {3.1.2} 50
execle() Execute a File. {3.1.2} 50
execlp() Execute a File. {3.1.2} 50
execv() Execute a File. {3.1.2} 50
execve() Execute a File. {3.1.2} 50
execvp() Execute a File. {3.1.2} 50
_exit() Terminate a Process. {3.2.2} 56
fcntl() File Control. {6.5.2} 117
<fcntl.h> Data Definitions for File Control Operations. {6.5.1} 116
fdopen() Open a Stream on a File Descriptor. {8.2.2} 146
fileno() Map a Stream Pointer to a File Descriptor. {8.2.1} 145
fork() Process Creation. {3.1.1} 49
fpathconf() Get Configurable Pathname Variables. {5.7.1} 105
fstat() Get File Status. {5.6.2} 99
getcwd() Working Directory Pathname. {5.2.2} 86
getegid() Get Real User, Effective User, Real Group, and Effective Group IDs. {4.2.1} 71

getenv() Environment Access. {4.6.1} 79
geteuid() Get Real User, Effective User, Real Group, and Effective Group IDs. {4.2.1} 71
getgid() Get Real User, Effective User, Real Group, and Effective Group IDs. {4.2.1} 71
getgrgid() Group Database Access. {9.2.1} 151
getgrnam() Group Database Access. {9.2.1} 151
getgroups() Get Supplementary Group IDs. {4.2.3} 73
getlogin() Get User Name. {4.2.4} 74
getpgrp() Get Process Group ID. {4.3.1} 75
getpid() Get Process and Parent Process IDs. {4.1.1} 71
getppid() Get Process and Parent Process IDs. {4.1.1} 71
getpwnam() User Database Access. {9.2.2} 152
getpwuid() User Database Access. {9.2.2} 152
getuid() Get Real User, Effective User, Real Group, and Effective Group IDs. {4.2.1} 71
group Group Database Access. {9.2.1} 151
<grp.h> Group Database Access. {9.2.1} 151
isatty() Determine Terminal Device Name. {4.7.2} 80
kill() Send a Signal to a Process. {3.3.2} 62
<limits.h> Numerical Limits. {2.9} 44
link() Link to a File. {5.3.4} 90
longjmp() Non-Local Jumps. {8.3.1} 150
lseek() Reposition Read/Write File Offset. {6.5.3} 120
main() Execute a File. {3.1.2} 50
mkdir() Make a Directory. {5.4.1} 92
mkfifo() Make a FIFO Special File. {5.4.2} 93
open() Open a File. {5.3.1} 87
opendir() Directory Operations. {5.1.2} 83
passwd User Database Access. {9.2.2} 152
pathconf() Get Configurable Pathname Variables. {5.7.1} 105
pause() Suspend Process Execution. {3.4.2} 68
pipe() Create an Inter-Process Channel. {6.1.1} 109
<pwd.h> User Database Access. {9.2.2} 152
read() Read from a File. {6.4.1} 111
readdir() Directory Operations. {5.1.2} 83
rename() Rename a File. {5.5.3} 96
rewinddir() Directory Operations. {5.1.2} 83
rmdir() Remove a Directory. {5.5.2} 95
setgid() Set User and Group IDs. {4.2.2} 72
setjmp() Non-Local Jumps. {8.3.1} 150
setlocale() Extensions to *setlocale*() Function. {8.1.2} 144
setpgid() Set Process Group ID for Job Control. {4.3.3} 75
setsid() Create Session and Set Process Group ID. {4.3.2} 75
setuid() Set User and Group IDs. {4.2.2} 72
sigaction() Examine and Change Signal Action. {3.3.4} 64
sigaddset() Manipulate Signal Sets. {3.3.3} 63

sigdelset() Manipulate Signal Sets. {3.3.3} 63
sigemptyset() Manipulate Signal Sets. {3.3.3} 63
sigfillset() Manipulate Signal Sets. {3.3.3} 63
sigismember() Manipulate Signal Sets. {3.3.3} 63
siglongjmp() Non-Local Jumps. {8.3.1} 150
<signal.h> Signal Concepts. {3.3.1} 57
sigpending() Examine Pending Signals. {3.3.6} 67
sigprocmask() Examine and Change Blocked Signals. {3.3.5} 66
sigsetjmp() Non-Local Jumps. {8.3.1} 150
sigsetops Manipulate Signal Sets. {3.3.3} 63
sigsuspend() Wait for a Signal. {3.3.7} 67
sleep() Delay Process Execution. {3.4.3} 69
stat() Get File Status. {5.6.2} 99
<stat.h> File Characteristics: Header and Data Structure. {5.6.1} 97
sysconf() Get Configurable System Variables. {4.8.1} 80
<sys/stat.h> File Characteristics: Header and Data Structure. {5.6.1} 97
<sys/types.h> Primitive System Data Types. {2.6} 40
<sys/wait.h> Wait for Process Termination. {3.2.1} 54
tar Extended `tar` Format. {10.1.1} 155
tcdrain() Line Control Functions. {7.2.2} 137
tcflow() Line Control Functions. {7.2.2} 137
tcflush() Line Control Functions. {7.2.2} 137
tcgetattr() Get and Set State. {7.2.1} 136
tcgetpgrp() Get Foreground Process Group ID. {7.2.3} 139
tcsendbreak() Line Control Functions. {7.2.2} 137
tcsetattr() Get and Set State. {7.2.1} 136
tcsetpgrp() Set Foreground Process Group ID. {7.2.4} 139
termios General Terminal Interface. {7.1} 123
<termios.h> Settable Parameters. {7.1.2} 129
time() Get System Time. {4.5.1} 77
times() Process Times. {4.5.2} 78
ttyname() Determine Terminal Device Name. {4.7.2} 80
<types.h> Primitive System Data Types. {2.6} 40
tzset() Set Time Zone. {8.3.2} 150
umask() Set File Creation Mask. {5.3.3} 90
uname() System Name. {4.4.1} 76
<unistd.h> Symbolic Constants. {2.10} 47
unlink() Remove Directory Entries. {5.5.1} 94
utime() Set File Access and Modification Times. {5.6.6} 103
<utsname.h> System Name. {4.4.1} 76
wait Wait for Process Termination. {3.2.1} 54
<wait.h> Wait for Process Termination. {3.2.1} 54
waitpid() Wait for Process Termination. {3.2.1} 54
write() Write to a File. {6.4.2} 113

Topical Index

/usr/group ... 4, 9, 166, 168, 171, 176, 178-180, 182-183, 195, 212, 223, 238, 245-246, 250, 256-258, 279
1003 ... 6, 166, 171-172, 177, 186, 273, 275
1003.0 ... **166**
1003.1 ... 3-4, 6, 9, 164, 171-172, 178, 182, 185, 252
1003.2 ... **164**, 4-5, 155, 164, 172, 217, 221, 273
1003.3 ... **166**, 5, 166, 172, 188-189
1003.4 ... **166**, 5, 166, 172, 273
1003.5 ... **166**, 5, 166, 172
1003.6 ... **166**, 5, 166, 172
4.1 ... 226
4.2 ... 4, 141, 179-180, 200-201, 212, 222-223, 225, 227, 231, 236-237, 241, 245, 252, 257-258, 261, 265, 272-273
4.3 ... 4, 141, 179-180, 194, 202-203, 208, 210, 212, 220-221, 223, 226-228, 230, 234, 236-238, 241, 245-246, 248, 252-253, 255, 262, 266, 269, 272-273, 275, 281
8-bit characters ... 157, 161
802.2 ... 167
802.3 ... 167
802.4 ... 167
<National Body> Conforming POSIX Application ... **25**
<sys/stat.h> File Modes ... **98**
abnormal termination ... 53, 57, 59, 220, 226
abnormal termination with actions ... 220
abort ... **141**, 53, 57, 141, 147, 149, 235, 272
abs ... **141**
absolute pathname ... **28**, 36, 86, 196, 203, 206
absolute value ... 54, 62, 231
access ... **100**
access control ... 30-31, 35, 58, 100, 123-124, 197, 200-201, 203-204, 229, 247, 256, 261, 264
access mechanism ... 35, 100, 155, 195, 203, 240-241, 256-257
access mode ... **28**, 31, 87, 91, 101, 116-119, 124, 257
access permissions ... 29-31, 34-35, 53, 91, 96, 100-101, 155, 160, 195, 247-248, 261
access time ... 36, 97, 103-104, 249, 257, 269
acos ... **141**
actime ... **104**, 249
active handle ... 147-148, 270
Ada ... 5, 166-167
Ada Language Bindings ... **166**
additional access control mechanism ... 35, 100
address space ... **28**, 27-28, 32-33
advisory record locking ... 118, 257
AFNOR ... 190
alarm ... **68**, 49-50, 52-53, 57, 61, 68-70, 230, 234-235, 238, 257
alarm clock ... 52
alarm requests ... 68
alternate access control mechanism ... 35, 100
amode ... 100-101
ANSI ... 26, 28, 163-164, 167-168, 174, 176, 178, 180, 185, 190, 278
append ... 114, 116, 141, 147, 160, 220, 269
Application Conformance ... **25**, 190
application developer ... 21-22, 175, 188, 204-206, 216, 239, 260, 267
Application Oriented ... **173**
application portability ... 3, 21, 23-24,

47, 172-173, 189, 205, 224, 231, 234-235
application programs ... 22-24, 41, 123, 164-165, 171, 173, 189, 192, 205, 214, 240-241, 260, 269-270, 281
application writer ... 147, 156, 172, 182, 195, 210-211, 215, 219, 229, 231, 234-235, 240, 244, 250, 253-256, 258-259, 269, 275
appropriate privileges ... **28**, 32, 35, 40, 48, 72-73, 91, 94, 101-104, 106, 155, 158, 161-162, 193, 203, 236, 248
archive ... 155-162, 180, 274-277
archive/interchange format ... 155, 162, 273
AREGTYPE ... 156
arg list ... 37
argc ... 50-51, 218
argument count ... 51, 96
argument list ... 37, 51, 53, 80, 100, 105, 219
argv ... 50-51, 218
ARG_MAX ... **45**, 37, 42, 45, 51, 53, 81, 219
array ... 29, 40, 42, 51, 73-74, 77, 79, 83, 86, 126, 129, 133-134, 157, 190, 209, 224, 236, 239, 241-242, 268
array size ... 83, 86, 190, 241-242
ASCII ... 142, 155, 157-161, 202, 218, 272, 274
asctime ... **142**
asin ... **141**
assert ... **141**
associated process group ... 28, 30-31, 123-124, 139-140, 227
asynchronous communications ports ... 123
asynchronous serial connection ... 132
asynchronous terminal ... 123, 132, 137, 258
AT&T ... 4, 171, 174, 178, 186, 193, 201, 212, 245, 250-251, 259, 280
atan ... **141**
atan2 ... **141**
atexit ... 183, 221-222
atof ... **141**
atoi ... **141**, 192-193
atol ... **141**
atomic ... 88, 217, 246, 253-254, 256
attention signal ... 57
background ... 28, 30, 58, 113, 115, 124, 132, 136, 139, 183, 197-199, 236, 261, 264
background process group ... **28**, 30, 58, 113, 115, 124, 136, 139, 198, 236, 261, 264
backward compatibility ... 158, 189, 211, 213, 224, 251, 263, 276-277
balanced trees ... 243
Balloting Group ... **176**, 9, 176, 183
bandwidth ... 205
Base by POSIX, Additions by X3J11 ... **182**
Base by X3J11, Additions by POSIX ... **182**
Base Documents ... **178**
Basic ... 167
baud ... 132, 134-135, 264
Baud Rate Functions ... **134**
Bell Laboratories ... 171, 179
Berkeley ... 179, 249, 258, 281
binary ... 22, 142, 155, 158, 174, 181, 232, 239, 249, 266, 269, 276
binary compatibility ... 158, 174, 232, 276
binary zero ... 155, 158
binding ... 3, 21, 26-27, 141, 166-167, 173, 181, 190-192, 225, 245
bit-encoded ... 98
bits ... 30, 35, 52, 55-56, 62, 65, 87, 90, 92-93, 98-99, 101-103, 118, 130-132, 137, 157-158, 208, 219, 221, 223, 232, 235, 244-245, 247-248, 252, 257, 263-264, 274
bitwise inclusive OR ... 87, 99, 101
BLKTYPE ... 156, 158
block special file ... **28**, 28-30, 88, 98, 156, 159, 161
blocked signal ... 59, 62-63, 66-67, 69, 118, 124-125, 133, 136, 219, 224, 226-234, 261
Bourne ... 210
break ... 21, 129-130, 137, 177, 188, 263, 267
brk ... 182
BRKINT ... 129-130
Broadly Implementable ... **174**
BSD ... 4, 179-180, 194, 197, 200-203, 208, 210, 212, 220-223, 225-228, 230-231, 233-234, 236-238, 241, 245-246, 248-253, 255-258, 261-

262, 265-266, 269, 272-273, 275, 281
bsearch ... **141**, 182
BSI ... 190
buffer ... 78, 84, 111-113, 127, 148-149, 188, 221, 242-243, 249, 252-253, 269-271
bug ... 241, 244, 276
bus ... 167
By Neither POSIX Nor X3J11 ... **182**
byte ... 28, 30, 32-33, 36-37, 42, 44-45, 51, 53, 74, 79, 83, 86, 111-115, 119, 121, 125-128, 130-131, 133, 142, 148-149, 155-162, 204-205, 215, 218, 237, 244, 249, 252-253, 255, 257, 262-263, 271, 274, 277-278
byte-oriented ... 159
C language binding ... 3, 26-27, 167
C Language Definitions ... **42**
C Language Library ... **141**
C language standard ... 3, 21, 26, 44, 53, 164, 173, 179, 182, 222, 230
C Language Standard ... **164**
C Shell ... 197-199
C Standard ... **28**, 26-28, 38, 40, 42-44, 53, 57, 60-61, 65-66, 99, 141, 144, 146, 150, 164, 173-174, 178, 180-183, 185-186, 188, 190, 192-193, 207-209, 211-214, 218-219, 221, 224, 229, 238, 246, 252, 256-258, 260
callable function ... 25, 61, 193
calloc ... **141**
canonical mode ... 45, 125, 127, 133, 262
Canonical Mode Input Processing ... **125**
CASE Services ... 167
catch ... 60-61, 66, 200, 207, 223-227, 233, 235, 264
caught signal ... 38, 52, 55, 57-58, 65-66, 68-69, 111, 118, 226, 233, 261
CBEMA ... 163, 167
CCITT ... 167
cc_t ... 129
ceil ... **141**
cfgetispeed ... **134**, 44, 61, 134-135
cfgetospeed ... **134**, 44, 61, 134-135
cfsetispeed ... **134**, 44, 61, 134-135, 264
cfsetospeed ... **134**, 44, 61, 134-135
Change Current Working Directory ... 85, 243
Change File Modes ... 101, 248
Change Owner and Group of File ... 102, 248
character ... **28**
character array ... 29, 42, 51, 74, 77, 79, 83, 86, 133, 157, 241-242, 268
character framing error ... 130
character pointers ... 51, 74, 79, 86, 268
character special ... 28-30, 34, 48, 57-58, 88, 98, 123, 125-128, 133-134, 156, 159-161, 197, 200, 232
character special file ... **28**, 28-30, 34, 88, 98, 128, 134, 156, 159, 161
character string ... 29, 41-42, 51, 79, 146, 157, 241, 268
CHAR_BIT ... 44
CHAR_MAX ... 44
CHAR_MIN ... 44, 213
chdir ... **85**, 61, 85-86, 157, 161, 243, 247
child process ... **28**, 32, 38, 49-50, 53-56, 58, 61, 65, 76, 78, 84, 119, 124, 145, 198, 216-217, 220-223, 228, 232, 237-238, 269
child times ... 78, 238
CHILD_MAX ... **45**, 81, 209, 214
chksum ... 156, 158
chmod ... **101**, 35, 44, 52-53, 61, 90, 93-94, 99, 101-103, 162, 248
chown ... **102**, 48, 61, 99, 102-103, 241, 248-249
chroot ... 203
CHRTYPE ... 156
clear errno ... 206, 242
clearenv ... 239
clearerr ... **141**, 221
CLK_TCK ... 28, 42, 78, 80-81, 234, 238, 240
CLOCAL ... 123, 128, 131-132, 260
clock tick ... **28**
clock_t ... 28, 42, 78, 238
close ... **111**, 52, 57, 61, 84, 89, 95, 110-112, 116-117, 119-120, 129, 132, 147-148, 183, 186, 188, 200, 224, 242, 251-252, 262, 270, 278
close file ... 52, 110-111, 116-117, 119, 129, 147-148, 188, 251-252, 262, 270
closedir ... **83**, 43, 83-85, 180, 242
Closing a Terminal Device File ... **129**
cmask ... 90, 245
cmd Values for fcntl ... **115**
collation ... 41

command ... 31, 41, 110, 118, 123, 136, 165, 184-185, 197-199, 211, 218-219, 221, 224-225, 231, 256, 258-260, 269, 273
command interpreter ... 31, 123, 197, 211, 218, 221, 225
commercial applications ... 22
common usage ... 26-27, 43, 192, 203, 208, 211-212, 218, 234, 265
Common Usage-Dependent Support ... 43
compatibility ... 4, 158, 161, 174, 189, 203, 205, 208-209, 211, 213, 224, 232, 238, 245, 250-251, 263, 275-277
compile time ... 31, 47, 237, 239, 266
Compile-Time Symbolic Constants ... **47**
compiler ... 31, 47, 181, 193, 211-213, 215, 229, 236-237, 239, 272
computer architecture ... 239
concurrency ... 22, 255-256
concurrent writes ... 255-256
configurable pathname variables ... 105, 239-240, 249
configurable system variables ... 80, 174, 239-240, 249
configuration ... 22, 47, 165, 190, 207, 240
Conformance ... **24**
conformance documentation ... 24, 26, 187-188
conformance test suite ... 168
conforming application ... 22-26, 31, 44, 47, 164-165, 175, 182, 188-190, 194, 201, 206, 210, 212, 214-215, 217-218, 225, 229, 234-235, 247, 250-253
Conforming Implementation ... 25, 189
conforming implementation ... **24**
Conforming POSIX Application ... **25**, 22-26, 44, 47, 182, 188, 190, 201, 214-215, 218, 225, 234-235, 251, 253
Conforming POSIX Application Using Extensions ... **26**, 23-24, 26, 190, 214-215, 251
conforming program ... 21-23, 26, 164-165, 174, 187, 190-191, 265, 269
conforming system ... 5, 24-25, 44, 142, 155, 161, 175, 182, 188, 191, 195, 207, 213, 247, 251, 253, 265, 276
connection ... 28, 30, 87, 123, 132, 135, 196, 222, 263
consensus ... 12, 175-177, 205, 224, 227, 251, 263, 278
constants ... 4, 25, 31, 45, 47-48, 57-58, 65, 74, 79-80, 101, 105, 115-116, 121, 123, 136, 138, 156, 172, 174, 184-185, 189, 216, 237, 239-240, 247, 250
constraining links ... 246
contiguous files ... 166, 273, 276-277
continue signal ... 59-62, 198-200, 217, 227-228, 261
control character ... 127-130, 133-134, 197, 215, 232, 261
Control Functions ... **136**
Control Modes ... **131**, 123, 128, 130-131, 260
Control Operations on Files ... 255
control-Z ... 197, 200
controlling process ... **28**, 30-33, 35, 54, 57-59, 63, 68-69, 74-75, 79, 88, 113, 115, 119, 123-124, 127-128, 130, 132-133, 136, 139-140, 197-203, 221-222, 228, 231, 236-237, 239, 244, 247, 256, 261, 264-265
controlling terminal ... **28**, 30-31, 57-58, 74-75, 79-80, 88, 113, 115-116, 123-124, 127-134, 136, 139-140, 165, 193, 197-202, 229, 232, 239, 244, 260-261, 263-265, 277
CONTTYPE ... 156
cooked mode ... 259
Coordinated Universal Time ... 34, 194, 265
core file ... **220**
core services ... 21, 173
corrupt ... 245
cos ... **141**
cosh ... **141**
covert channel ... 205, 232
cpio ... **159**, 155, 159-161, 180, 273-277
cpio Archive Entry ... **160**
cpio c_mode Field ... **161**
CPU time ... 78
CREAD ... 131-132
creat ... **90**
creat function ... 32-33, 49-50, 75-76, 87, 89-91, 109, 118-119, 124, 147, 157, 161, 191-192, 198, 227, 236, 245, 247, 256, 270

create new file ... 90-92, 119, 124, 147, 241, 245
Create Session and Set Process Group ID ... **75**
created directory ... 87, 89, 92-93, 95, 106, 220, 241
csh ... 199
CSIZE ... 131
CSMA/CD ... 167
CSTOPB ... 131-132
ctermid ... **79**, 79-80, 182, 239
ctime ... **142**, 142-143, 150
current directory ... 29, 35-36, 41, 52, 84-86, 95, 106, 210, 220, 243, 246, 249
current working directory ... **29**, 35-36, 41, 52, 85-86, 95, 210, 220, 243, 246, 249
curses ... 258
cuserid ... **74**, 152-153, 182
c_cc ... **133**, 126, 129, 133-134
c_cflag ... **131**, 128-129, 131, 264
c_dev ... 160, 275
c_filedata ... 160-161
c_filesize ... 160-161
c_gid ... 160
c_iflag ... **129**, 125, 129
c_ino ... 160, 275, 277
C_IRGRP ... 161
C_IROTH ... 161
C_IRUSR ... 161
C_ISBLK ... 161
C_ISCHR ... 161
C_ISCTG ... 161
C_ISDIR ... 161
C_ISFIFO ... 161
C_ISGID ... 161
C_ISLNK ... 161
C_ISREG ... 161
C_ISSOCK ... 161
C_ISUID ... 161
C_ISVTX ... 161
C_IWGRP ... 161
C_IWOTH ... 161
C_IWUSR ... 161
C_IXGRP ... 161
C_IXOTH ... 161
C_IXUSR ... 161
c_lflag ... **132**, 125, 129, 132
c_magic ... 160
c_mode ... 155, 160-161, 275, 277
c_mtime ... 160
c_name ... 160-161
c_namesize ... 160-161, 277
c_nlink ... 160
c_oflag ... **131**, 127, 129, 131
c_rdev ... 160, 275
c_uid ... 160
daemon ... 165, 222
Data Definitions for File Control Operations ... **116**, 31, 116, 251, 255
Data Interchange Format ... 155, 273
data structure ... 36, 97, 161, 188, 191, 202, 207, 217, 224, 229-230, 242, 247, 264
Data Types ... 40, 184, 208, 247
database ... 22, 41, 74, 151-152, 165, 168, 196, 201, 256-257, 272-273
database access ... 151-152, 273
Database Standards ... **168**
Date and Time ... 142
date/time ... 41, 267
daylight saving time ... 265
de facto standard ... 190, 277
deadlock ... 38, 119-120, 257
debugging ... 221
decimal ... 143, 156, 194
decrement ... 94
default ... 52, 57-60, 124, 133, 138, 142-144, 150-151, 179, 199-200, 204, 211-212, 219, 224, 226-229, 232, 252, 266, 268-269
default action ... 52, 59-60, 124, 200, 219, 226-229, 232
default shell ... 199-200
delay ... 68-69, 88, 113, 115-116, 230, 234-235
delay process execution ... 69, 230, 234
deliver ... 54, 56, 59-63, 65-70, 217, 224-228, 230-231, 233
Department of Defense ... 168
Determine Terminal Device Name ... 80, 239
device ... **29**, 28-30, 34, 39-40, 80, 88, 97, 112, 115, 121, 123-125, 127-131, 134-135, 138, 147, 158-159, 167, 180, 183, 194, 201, 207, 215, 230, 239, 244, 246, 249, 252-253, 256, 258-260, 262-264, 271, 274, 276
device number ... 97, 125, 130, 158-159, 194, 215, 274
Device- and Class-Specific Functions ...

123, 180, 183, 258
device-dependent ... 258
devmajor ... 156, 159
devminor ... 156, 159
dev_t ... **40**, 97, 208
Diagnostics ... 141
digits ... 142-143, 158-160, 265
dir.h ... 241
directory ... **29**
directory ... **83**
directory access ... 29, 35, 89, 91, 96, 241, 243
directory entry ... **29**, 31, 84, 91-93, 95-96, 194, 207, 242, 245
directory format ... 83, 194, 241, 275
directory level ... 240, 249
Directory Operations ... **83**, 49, 83, 174, 180, 194, 242
directory routines ... 194
directory search permission ... 35, 53, 89, 91-92, 94-95, 97-99, 244
directory stream ... 49, 56, 84-85, 242-243
dirent ... 83, 85, 241-242
dirent.h ... **83**, 43, 83-85, 241
dirname ... 83-85
DIRTYPE ... 156, 158
disconnect ... 57, 124, 128-129, 135, 222, 262
document ... 4-5, 12, 23-27, 59, 67, 163-166, 168, 171-172, 175-182, 186-187, 194, 196, 204, 212, 224-225, 247, 273, 275-277
Document Indexes ... **186**
Documentation ... **24**, 51, 59, 77
domain ... 27, 38, 196, 218, 222, 266, 272
dominate ... 247
dot ... **29**, 29-31, 36, 39, 84, 95, 97, 189, 194, 210, 246
dot-dot ... **29**, 29-31, 36, 39, 84, 95, 97, 189, 194, 205, 246
double initial slash ... 205
dup ... **110**, 61, 89, 100, 110-111, 113, 115, 121, 147, 251, 256, 269
dup2 ... **110**, 61, 110, 251
Duplicate an Open File Descriptor ... 110, 251
d_name ... 83, 241-242
E2BIG ... **37**, 53, 219
EACCES ... **37**, 53, 76, 85-86, 89, 91-95, 97, 100-104, 106, 120, 237, 244
eaccess ... 244, 247
EAGAIN ... **37**, 50, 112-115, 120, 125, 217, 250, 253-255
EBADF ... **37**, 85, 100, 107, 110-111, 113, 115, 120-121, 137-140, 251, 253, 255
EBUSY ... **37**, 94-95, 97, 207, 246
ECHILD ... **38**, 56
ECHO ... 132-133
ECHOE ... 132, 263
ECHOK ... 132-133, 263
ECHONL ... 132-133, 263
EDEADLK ... **38**, 119-120, 257
EDOM ... **38**, 208
EEXIST ... **38**, 89, 91-93, 95, 97, 245-246
EFAULT ... **38**, 206-207, 219
EFBIG ... **38**, 115
effective group ID ... **29**, 29-31, 33-34, 48, 52, 71-72, 87, 92, 99-100, 102-103, 219, 235, 241, 248
effective user ID ... **29**, 29-31, 33-34, 52, 62, 71-72, 74, 87, 92-93, 99-100, 102, 230, 235
Eighth Edition ... 253, 255
EINTR ... **38**, 56, 68-69, 89, 111-115, 118, 120, 138, 184, 207, 217, 219, 223, 230, 252
EINVAL ... **38**, 56, 63-64, 66-67, 73, 76, 81, 86, 97, 101, 103, 106-107, 120-121, 137-138, 140, 233, 250, 258
EIO ... **38**, 113, 115, 124, 128
EISDIR ... **38**, 89, 97
elapsed real time ... 69, 78
EMFILE ... **38**, 85, 89, 109-110, 120
EMLINK ... **38**, 91-92
empty ... **29**
empty directory ... 29, 39, 84, 86, 92, 95-96
empty pipe ... 112, 254
empty string ... **29**, 31, 39, 53, 80, 86, 89, 91-95, 97, 100-104, 106, 144-145
ENAMETOOLONG ... **38**, 53, 85-86, 89, 91-95, 97, 100-104, 106, 219, 245-246
encoded ... 159, 202, 204, 221, 245, 264, 272
encoded password ... 272
end-of-archive ... 155, 278
end-of-directory ... 242
end-of-file ... 112, 125, 127-128, 142, 146, 162, 250-251, 255, 277-279

end-of-medium ... 278-279
endgrent ... 272
endpwent ... 272
ENFILE ... **39**, 85, 89, 109
ENODEV ... **39**
ENOENT ... **39**, 53, 85-86, 89, 91-95, 97, 100-104, 106, 246
ENOEXEC ... **39**, 53, 218-219
ENOLCK ... **39**, 120, 257
ENOMEM ... **39**, 50, 53, 207, 217, 271
ENOSPC ... **39**, 89, 91-93, 97, 115
ENOSYS ... **39**, 76, 139-140, 187
ENOTDIR ... **39**, 53, 85-86, 89, 91-95, 97, 100-104, 106
ENOTEMPTY ... **39**, 95, 97, 245-246
ENOTTY ... **39**, 137-140, 207
environ ... **50**, 40, 42, 50-51, 79, 219
environment ... 3-4, 21, 24, 37, 40-45, 51, 53, 69, 71, 79, 142, 144-145, 150, 163, 165, 168, 171, 173, 181-182, 184-185, 188, 191, 193-194, 203, 210-213, 218-219, 235, 239-240, 257-258, 262, 265-270
environment access ... 79, 203, 219, 239
Environment Description ... **40**, 4, 40, 51, 53, 79, 182, 210
environment list ... 37, 51, 53, 79, 219
environment strings ... 40-41, 51, 79, 144-145, 268-269
Environment Variables ... **40**, 142, 184, 239
envp ... 42, 50-51, 218-219
ENXIO ... **39**, 89
EOF ... 47, 118, 125-128, 133, 148-149, 271
EOL ... 125-128, 133
EPERM ... **39**, 63, 73, 75-76, 91, 94, 102-104, 140, 232
EPIPE ... **40**, 115, 207
Epoch ... **29**, 34, 77, 99, 104, 142, 185, 194-195, 265
ERANGE ... **40**, 86, 208, 244
ERASE ... 126-128, 132-133, 262
ERASE character ... 126-127, 132-133, 262
EROFS ... **40**, 89, 91, 93, 95-97, 101-104, 207-208
errno ... **37**, 4, 37, 50, 52-53, 56, 63-69, 72-73, 75-78, 81, 85-86, 89, 91-95, 97, 100-104, 106, 109-115, 118-121, 124-125, 128, 136-140, 145-146, 149, 182, 184, 186, 203, 206-207, 223, 230, 242, 246, 250, 254-255, 257, 270
errno.h ... **37**, 211, 246
error code ... 92, 95, 97, 225
error number ... 4, 37, 89, 113, 115, 174, 182, 184, 206, 208, 230, 253
Error Numbers ... **37**
ESPIPE ... **40**, 121
ESRCH ... **40**, 63, 76, 231-232
ETXTBSY ... 219
event notification ... 166, 224
Examine and Change Blocked Signals ... 66, 233
Examine and Change Signal Action ... 64, 200, 225, 233
Examine Pending Signals ... 67, 233
exclusive lock ... 116, 118-120, 257
EXDEV ... **40**, 91, 97
exec ... **50**, 29, 33, 39, 42, 44-45, 50-53, 60, 65, 68, 71, 73, 75-76, 79, 84, 89, 94, 99, 110-111, 116-117, 120, 147-148, 179, 182, 186, 198, 216, 218-220, 235, 237, 242, 247, 252, 256, 269-270
exec family ... 50, 84, 116-117
exec functions ... 44-45, 50-53, 65, 84, 116-117, 147-148, 186, 198, 216, 218-220, 242, 247, 270
execl ... **50**
execle ... **50**, 53, 61, 218
execlp ... **50**, 50-51, 53, 218
executable binary form ... 249
executable form ... 240, 249
execute access bits ... 35, 248
execute by group ... 161, 199
execute by others ... 34, 61, 161, 248
execute by owner ... 161
execute file ... 35, 39, 50, 88, 98-101, 111, 147, 218, 247-248, 257
execute mode ... 257
execute permission ... 35, 39, 47, 98-101, 247-248
execution environment ... 181, 188
execution time ... 48, 69, 78, 227, 230, 239, 266
Execution-Time Symbolic Constants ... **48**
execv ... **50**
execve ... **50**, 50-51, 53, 61, 218
execvp ... **50**, 50-51, 53, 218

exit ... **141**, 53, 55-57, 141-142, 147, 149, 183, 201, 216, 221-223, 229, 252, 270, 272
exit status code ... 56
EXIT_SUCCESS ... 142, 222
exp ... **141**
extended cpio ... 155, 159, 180, 273, 277
Extended cpio Format ... **159**
extended function ... 35, 39, 121, 157, 161, 175, 229-230, 248, 258
extended security ... 35, 63, 203
extended security controls ... **35**, 63, 203
extended tar ... 155, 175, 180, 273, 276
Extended tar Format ... **155**
extension ... 4-5, 23-24, 26, 37, 41, 43, 55, 141-142, 144, 150, 159, 165-166, 173, 175, 188, 190-191, 193-194, 196, 202-203, 208, 210, 213-215, 217, 220-221, 224, 228, 230-232, 238-239, 245, 247-248, 251, 253, 263-265, 267, 270, 273, 275-276
Extensions to setlocale() Function ... **144**
Extensions to Time Functions ... **142**
external variable ... 4, 37, 40, 51, 143, 150, 181
fabs ... **141**
Fast File System ... 249
fclose ... **141**, 147-149, 183, 271
fcntl ... **117**, 44, 49-51, 53, 61, 89, 100, 109-111, 113, 115-121, 125, 147, 157, 161, 208, 251, 254-258, 260, 269
fcntl Return Values ... **119**
fcntl.h ... **116**, 44, 51, 84, 87, 89-90, 116-118, 120
fdopen ... **146**, 44, 146-147, 182, 269
FD_CLOEXEC ... 51, 84, 116-117, 147
feature ... **29**, 23-24, 29, 40, 42-44, 164, 171-172, 174, 177-178, 187, 190-191, 197, 202, 208, 211-213, 217, 220, 223-224, 231, 240, 248-250, 256-258, 260, 262-263, 273-274
feature test macro ... 29, 40, 42-43, 208, 211-213
Federal Information Processing Standards ... 168
feof ... **141**, 147
ferror ... **141**
fflush ... **141**, 147-149, 270-271
fgetc ... **141**, 147, 149, 271
fgetpos ... 209, 258
fgets ... **141**, 147, 149, 271
FIFO ... 29-30, 32, 40, 87-89, 93, 98, 106, 111-112, 114-115, 121, 156, 159, 161, 179, 195, 202, 241, 244-245, 253-254
FIFO special file ... **29**, 29-30, 32, 93, 111-112, 156, 179
FIFOTYPE ... 156, 158
fildes ... 80, 99-100, 105-107, 109-115, 117-118, 120-121, 136-140, 146, 250-251, 254
fildes2 ... 110, 251
file ... **29**
file access control ... 31, 35, 100, 201, 203, 247, 256, 261
file access mask ... **116**
file access modes ... **116**
File Access Modes Used For open and fcntl ... **116**
file access permissions ... **35**, 29-31, 34-35, 53, 91, 96, 100-101, 155, 160, 247, 261
file accessibility ... 100, 247
File Characteristics ... 97, 246-247
file control operations ... 31, 39, 116, 247, 251, 255
file creation mask ... 52, 87, 90, 92-93, 245
file description ... **29**, 29-31, 49, 51, 87, 109, 111, 117-118, 121, 136, 145-149, 158, 201, 229, 242, 251, 255-256, 270-271, 275
file descriptor ... **30**, 30-32, 37-38, 49, 51, 56, 80, 84-85, 87, 89, 100, 102, 105-107, 109-111, 113-121, 124, 136-140, 145-149, 190, 201, 239, 242, 244, 247, 249-252, 255-256, 269-270
File Descriptor Flags Used For fcntl ... **116**
File Descriptor Manipulation ... 251
file group class ... **30**, 35, 98
file hierarchy ... **36**, 32, 36, 155, 157, 161, 196, 203, 245, 274-275
file mode ... **30**, 28, 30-31, 52, 87, 90-93, 97-98, 101-103, 116-119, 125, 130, 146-147, 155, 157-158, 181, 244, 248, 257, 262, 269, 274, 276
file mode creation mask ... 52, 87, 90, 92-93

file name ... 29-30, 32, 38, 45, 92, 96-97, 101, 105-107, 157-161, 185, 218, 242, 244, 246, 249-250, 275-277
file name length ... 45, 161, 242
file offset ... **30**, 30-31, 47, 87, 112-114, 119-121, 146-149, 253, 255, 257-258
file other class ... **30**, 35, 98-99
file owner class ... **30**, 35, 98
file owners ... 30, 35, 40, 52, 97-99, 101-104, 157-159, 248, 274
file permission ... 29-31, 34-35, 39, 53, 87, 89-93, 96, 98-101, 104, 155, 157-158, 160-161, 235, 244, 247, 249, 261
file permission bits ... **30**, 35, 87, 90, 92-93, 98-99, 101, 157-158
file pointer ... 80, 104, 118, 145, 252, 269, 271
file record locking ... 39, 111, 118, 248, 256-257
File Removal ... 245-246
file serial number ... **30**, 97, 158-159, 196, 241
file size ... 38, 97, 115, 121, 155-156, 158, 190, 242, 258, 275-277
file space ... 30, 39, 92, 94, 96, 111, 114-115, 157, 188, 190, 253
File Status ... 99, 116, 247
file status ... **116**
file status flags ... 87, 113-114, 116-119, 255
File Status Flags Used For open and fcntl ... **116**
file system ... **30**, 33, 36, 39-40, 45, 83, 85, 87, 89, 91-93, 95-97, 100-104, 109, 125, 155, 157, 159, 161, 165, 179, 185, 188-189, 194, 196, 201, 203-205, 207-208, 216, 219, 240-241, 245-250, 252-253, 267, 271-278
file table entry ... 201
file times update ... **36**, 52, 99-100, 205, 257, 269
file types ... 28-29, 31, 36, 40, 53, 84, 87-88, 97-98, 106, 117-120, 145-146, 148, 155, 158-161, 181, 195, 208-209, 241, 244, 248-249, 253, 257-258, 269, 274-277
filename ... **30**, 29-30, 32, 36, 41-42, 44, 51, 83, 88, 105, 157, 161, 182-183, 194-195, 202, 204-205, 210, 218-220, 241, 243, 275
filename portability ... **36**, 183, 202, 204, 210
fileno ... **145**, 44, 145-147, 182, 201
Files and Directories ... 83, 241
FIPS ... 168, 272
first-in-first-out ... 29, 109, 256
flag ... 51, 54-55, 60, 64-65, 84, 87, 109, 112-119, 123, 125, 127-128, 130, 132, 146, 162, 198, 200, 209, 220-221, 231, 233, 244-245, 250-251, 254-256, 263-264
floating point values ... 214
flock ... 118-119, 257
flock Structure ... **118**
floor ... **141**
flow control ... 130, 215
flush ... 130, 132-133, 137-138, 221, 253, 271
fmod ... **141**
fold ... 42, 204-205
fopen ... **141**, 146-149, 196, 269, 271
foreground ... 28, 30-31, 57, 123-124, 127-128, 130, 133, 136, 139, 197-201, 222, 236, 261, 265
foreground process group ... **30**, 28, 30-31, 57, 123-124, 127-128, 130, 133, 139, 197-201, 222, 236, 261, 265
foreground process group ID ... **31**, 139, 198, 200, 236, 265
foreground/background checks ... 201
fork ... **49**, 32-33, 49-50, 53, 56, 61, 68, 71, 75, 79, 84, 111, 119, 124, 145, 147-148, 198, 209, 216-217, 220, 236-237, 242, 255-256, 269-270
Format of Directory Entries ... 83, 194, 241
format-creating utility ... 155, 157-158, 161, 278-279
format-reading utility ... 155, 157-158, 161-162, 276-279
FORTRAN ... 5, 165, 167, 173, 190-191
fpathconf ... **105**, 48, 105-106, 250
fprintf ... **141**, 258
fputc ... **141**, 147, 149, 271
fputs ... **141**, 147, 149, 271
framing ... 130
fread ... **141**, 149, 271
free ... **141**
free function ... 39, 182, 223, 230, 243

freopen ... **141**, 147-148, 271
frexp ... **141**
fscanf ... **141**, 149, 271
fseek ... **141**, 141-142, 148-149, 271
fsetpos ... 209, 258
fstat ... **99**, 35-36, 44, 61, 97, 99-100, 247
FTAM ... 167
ftell ... **141**, 149, 271
Full Use ... 176, 179-180, 183, 248, 274
full-duplex mode ... 125
function address ... 27, 32-33, 60, 164, 193, 207, 221, 230, 265
function argument ... 27, 38, 54, 63-67, 69, 73, 77, 84-87, 96, 100, 104, 117-118, 120-121, 150, 218, 221-222, 229, 232, 243-244, 247, 250, 252, 259-260
function descriptions ... 27, 37, 49, 87, 101, 109, 118, 121, 136, 146-147, 192, 206, 216, 270
fwrite ... **141**, 149, 258, 271
F_DUPFD ... 110, 115, 117, 119-120
F_GETFD ... 115, 117, 119
F_GETFL ... 115, 117, 119
F_GETLK ... 115, 118-120
F_OK ... 47, 101
F_RDLCK ... 116, 118, 120
F_SETFD ... 115, 117, 119, 256
F_SETFL ... 115, 118-119, 256
F_SETLK ... 115, 118-120, 257
F_SETLKW ... 115, 118-120, 257
F_UNLCK ... 116, 118
F_WRLCK ... 116, 118, 120
general concepts ... 35, 193, 203, 211, 274
General File Creation ... 244
general terminal interface ... 28, 34, 123, 136, 259-260, 264
General Terminal Interface ... **123**
General Terms ... **28**, 4, 28, 183-184, 193
General Utilities ... 141
Generate Terminal Pathname ... 79, 239
Get Foreground Process Group ID ... **139**
getc ... **141**, 149, 271
getchar ... **141**, 149, 271
getcwd ... **86**, 243-244
getegid ... **71**, 61, 71-72, 236
getenv ... **79**, 141, 182, 219
geteuid ... **71**, 61, 71-72
getgid ... **71**, 61, 71-72
getgrent ... 272
getgrgid ... **151**, 44, 151-152, 272
getgrnam ... **151**, 44, 151-152, 272
getgroups ... **73**, 61, 73, 180, 236, 244-245
gethostid ... 238
gethostname ... 238
getlogin ... **74**, 152-153
getpgrp ... **75**, 61, 75-76, 199, 236
getpgrp2 ... 61, 75-76, 199, 236
getpid ... **71**, 61, 63, 71, 75, 230
getppid ... **71**, 61, 71
getpwent ... 272
getpwnam ... **152**, 44, 74-75, 152-153, 272
getpwuid ... **152**, 44, 74-75, 152-153, 272
gets ... **141**
gettimeofday ... 238
getty ... 135, 190, 236, 256, 260
getuid ... **71**, 61, 71-74, 209, 230
getwd ... 243
gid ... 72-73, 151, 156-159, 161, 248
gid_t ... **40**, 31, 40, 71-73, 97, 102, 151-152, 208-209
GKS ... 167
Global Externals ... 184
GMT ... 194
gmtime ... **142**, 195
gname ... 156-157, 159
grandchildren ... 220
graphics ... 22, 28, 165, 167, 191, 202
Graphics Standards ... **167**
Greenwich Mean Time ... 194
group ... **151**
group database ... **151**, 272-273
group database access ... 151, 273
group databases ... 151, 272-273
group file ... 29-31, 34-35, 48, 52, 87, 97-99, 101-103, 140, 155-159, 181, 196, 199, 216, 241, 244, 247-248, 261, 272
group ID ... **31**, 29-34, 40, 44, 48-49, 52, 62, 71-73, 75-76, 87, 92-93, 97, 99-100, 102-103, 139, 151-152, 158, 160, 179, 198, 200, 203, 208, 219, 235-237, 241, 248, 265
group name ... 5, 151, 185, 221, 251
group Structure ... **151**
grouplist ... 73
grp.h ... **151**, 44, 151, 159, 162, 209
gr_gid ... 151
gr_mem ... 151

gr_name ... 151
hang up ... 131, 135
hard link ... 273, 277
hardware ... 28, 34, 39, 53, 57, 59, 77, 130-132, 135-136, 171-172, 174, 177, 183, 188, 190, 196, 207, 214, 252-253, 255, 258, 260, 263, 274
hardware control ... 130-132, 260
header ... 4, 21, 24, 27, 29, 37-38, 40-43, 47-48, 54, 57, 64-66, 77, 83-84, 97, 100, 104, 116-118, 121, 129, 155-161, 182, 184, 186, 192, 208-209, 211-213, 238, 240-241, 247, 275, 277
Header and Data Structure ... 97, 247
header block ... 155-158, 161
header prototype ... 43
Headers and Function Prototypes ... **43**
hertz ... 240
hierarchy ... 32, 36, 155, 157, 161, 165, 196, 203, 245, 274-275
historical implementation ... 173-175, 178, 181-182, 189, 194-196, 201-202, 204-205, 207, 209-210, 215, 220-221, 223, 238, 241-243, 245-248, 251, 255-260, 263-264, 266-267, 271, 279
Historical Implementations ... **178**
historical reasons ... 190, 246, 251, 267
historical term ... 181, 201, 239, 259, 264
historical usage ... 4, 175, 264-265
history ... 165, 280
HOME ... **41**
home directory ... 41
HUPCL ... 129, 131-132
i-node ... 241, 277
I/O ... 22, 30-31, 39, 83, 87, 109, 126, 129, 148, 156, 159, 165-166, 190-191, 197-200, 230, 250, 253, 257, 270
ICANON ... 127-128, 132-134, 263
ICRNL ... 128-130
Identifier Index ... **186**
IEEE ... 3-6, 9, 11-12, 21, 24, 26-27, 32, 47, 61, 141-142, 144, 161, 163-167, 171-173, 175-178, 183-186, 193, 221, 246, 273, 279
IEEE Balloting Process ... **176**
IEEE Consensus Process ... **175**
IEEE Standards Board ... 5, 9, 11-12, 177
IEXTEN ... 128, 132-133
IFS ... 211
IGNBRK ... 129-130
IGNCR ... 128-130
IGNPAR ... 129-130
implementation characteristics ... 189-190, 216
implementation conformance ... 21-26, 31-32, 44, 47, 141, 164, 174-175, 177, 182, 187-190, 194, 200-201, 207, 210, 213-214, 217-218, 223-225, 227, 229, 231, 245, 250-251, 266
implementation details ... 174, 177, 203, 254, 266
implementation-defined ... **23**, 23-25, 28, 31-33, 35, 37-38, 40, 48, 56-57, 59, 62-63, 67, 74, 77-80, 99-101, 103, 113, 115, 123-124, 128, 131-133, 135, 137, 142, 144, 146, 150, 152-153, 155, 158-160, 162, 165, 187, 194-196, 201, 204, 206, 220-221, 224, 226, 228, 231-232, 234, 243-245, 258, 266-269, 271, 273, 275, 277-278
implementation-dependent ... 210, 219, 222, 225, 241, 269
implementation-independent ... 210, 228, 241, 274
implementation-specific ... 77, 245, 248, 258
implementor ... 3, 26-27, 173, 186, 193, 204, 208, 212-214, 222-223, 227, 231, 237, 245, 248, 260, 263, 272-275
inclusive OR ... 87, 99, 101
incomplete pathname ... 196
Industry Open Systems Publications ... **168**
inheritance of process attributes ... 52
inherits ... 49-50, 52, 119, 124, 198, 200-201, 217, 222, 255-256, 263, 269
init ... 200, 222, 231
initial user program ... 151-152
initial working directory ... 41, 151-152
initial working directory field ... 151
initialization ... 49, 51, 59, 63-64, 92-93, 150, 158, 188, 200, 206, 217, 232, 240
INLCR ... 129-130
inline ... 4, 193

ino_t ... **40**, 97
INPCK ... 129-130
Input and Output ... 83, 109, 245, 250, 252, 255
Input and Output Primitives ... **109**, 83, 109, 245, 250, 255
input character ... 123, 125-128, 130-134, 261-263
input control value ... 129, 131
input modes ... 45, 125-127, 129-130, 133, 138, 215, 262-263
Input Modes ... **129**
input parity checking ... 129-130
input processing ... 45, 123-127, 129-130, 132-133, 261-262
Input Processing and Reading Data ... **125**
input queue ... 44-45, 125-126, 130-131, 133, 215, 261-262
Input/Output ... 141
Institutional Representatives ... **176**, 176-177
int ... 43, 50, 54, 56, 60, 62-64, 66-68, 72-73, 75-76, 80, 83, 85-87, 90, 92-96, 99-103, 105, 109-111, 113, 117-118, 120, 134, 136-137, 139, 144-146, 150, 192, 209, 223, 229, 234-235, 242, 252, 254, 260, 267
integer ... 30-34, 47, 55, 89, 109, 113, 115, 117, 145, 158, 195, 221, 224, 238, 252
integral ... 55, 57, 129, 135, 208-209
inter-process communication ... 179
inter-process signals ... 270
Interactions of Other FILE-Type C Functions ... **146**
intercharacter timer ... 262
Interface Characteristics ... **123**, 113, 123, 260
Interface, Not Implementation ... **174**
international applications ... 171, 240, 267
interpreter ... 31, 123, 197, 211, 218, 221, 225
interrupted call ... 38, 61, 69, 223, 227, 230
interrupted operation ... 89, 113, 115
interval ... 28, 59, 217, 226
INTR ... 127-128, 133, 224
INT_MAX ... 44, 112, 114, 252
INT_MIN ... 44
iocntl ... 259
ioctl ... 207, 258-260, 265
IO_ERR ... 252
isalnum ... **141**
isalpha ... **141**
isascii ... 182
isatty ... **80**, 182, 206
iscntrl ... **141**
isdigit ... **141**
isgraph ... **141**
ISIG ... 127-128, 132-133, 200
islower ... **141**
ISO ... 24-26, 34, 163, 167, 185, 205, 269
ISO Conforming POSIX Application ... **25**
ISO member body ... 24-25
isprint ... **141**
ispunct ... **141**
isspace ... **141**
ISTRIP ... 129-130, 263
ISUID ... 274
isupper ... **141**
isxdigit ... **141**
IXOFF ... 128-130, 133, 138, 215
IXON ... 128-130, 133
job access control ... 197, 201, 261
job control ... **31**, 25, 31, 47, 54, 57-58, 75, 113, 115, 123-124, 127-128, 133, 136, 179, 187, 197-202, 207, 220-222, 225, 228, 231-232, 236-237, 239, 258, 261
job control shell ... 197-201, 222, 225, 228, 231, 236-237, 261
job control signals ... 57-58, 133, 197-200, 202, 225, 228, 232, 236, 261
Job Control Signals ... **58**
Julian ... 7, 143, 265, 267
kernel ... 167, 195-196, 201, 205, 222, 231, 281
kill ... **62**, 38, 44, 50, 57, 59-63, 66, 69, 71, 75, 126-128, 132-133, 181, 183, 209, 224, 226-233, 260-261
Korn shell ... 210
LAN ... 247
LANG ... **41**, 144-145
Language Binding ... 26-27, 141, 173
Language Standards ... **167**, 164, 167
Language-Dependent Services for the C Programming Language ... **26**
Language-Dependent Support ... 26, 43
Language-Specific Services for the C

Programming Language ... **141**
LC_ALL ... 144, 267-268
LC_COLLATE ... **41**, 144, 267
LC_CTYPE ... **41**, 144, 267
LC_MONETARY ... **41**, 144, 267
LC_NUMERIC ... **41**, 144, 267
LC_TIME ... **41**, 144, 267
ldexp ... **141**
leap seconds ... 194-195
library functions ... 27, 141, 174, 180, 192, 220, 230, 233
library routine ... 203, 207, 220, 225, 233
lifetime ... 29, 31-34, 81, 226, 232, 240, 250
limits.h ... **44**, 24-25, 44-46, 80-81, 105-106, 182, 185, 188, 190, 213-215, 225, 239, 249-250
line control functions ... 137, 264
line speeds ... 135, 259
linefeed ... 263, 271
link ... **90**, 29, 31, 38, 40, 44-45, 61, 90-92, 94-97, 99, 111, 156, 158-160, 162, 167, 194, 203, 208, 245-246, 273, 275-277
link count ... **31**, 38, 44-45, 91-92, 94-96, 111, 208
link definition ... **29**
Link Layer Control ... 167
linking across file systems ... 91
linkname ... 156-158, 276
LINK_MAX ... **45**, 38, 45, 91-92, 105
LNKTYPE ... 156, 158
Local Area Network ... 247
local control value ... 132-133
local file system ... 157, 161, 275
Local Modes ... **132**, 125, 127, 132, 198
locale ... **144**, 41, 125, 127, 129, 132-133, 142-145, 157, 159, 161, 198, 247, 263, 265-269, 275
locale.h ... 41, 142, 144, 268
localtime ... **142**, 142-143, 150, 195, 265
lock ... 7, 38-39, 49, 51, 111, 115-116, 118-120, 166, 248, 256-257
locked region ... 118-120, 256-257
lockf ... 38-39, 49, 51, 111, 115-116, 118-120, 166, 248, 256-257
locking process ... 111, 118-120, 248, 256-257
locking requests ... 118-120, 257
log ... **141**
log10 ... **141**
logical device ... 201
login account ... 41
login name ... 41, 74, 151-152, 244
login shell ... 197-198, 200
LOGNAME ... **41**, 210
logout ... 201
longjmp ... **141**, 69, 141, 150, 183, 207, 230, 235, 272
longjmp ... **150**, 69, 141, 150, 183, 207, 230, 235, 272
LONG_MAX ... 44
LONG_MIN ... 44
lowercase ... 32, 42, 142, 183, 185, 204, 210
lread ... 209, 252
lseek ... **120**, 29, 40, 47, 61, 89, 113, 115, 120-121, 142, 147-149, 190, 208-209, 216, 257-258, 271
lvalue ... 241
lwrite ... 252
L_ctermid ... 79, 239
L_cuserid ... 74, 236
l_len ... 118-119
l_pid ... 118-119
l_start ... 118-119, 257
l_type ... 116, 118
l_type Values For Record Locking With fcntl ... **116**
l_whence ... 118-119
machine ... **77**
macro ... 27, 29, 40, 42-43, 55, 98, 142, 150, 184-186, 192, 208, 211-213, 221, 272
magic ... 156-157, 159-160, 209, 275
magic bytes ... 157, 160
magic cookie ... 209
magic field ... 157, 159
mail ... 175, 211
main ... **50**, 5, 50-51, 53, 55, 60, 183, 207, 209, 218-219, 222, 255
make directory ... 92, 245
malloc ... **141**, 182, 230, 243, 271
mandatory locks ... 248, 257
manipulate signal sets ... 63, 232
marked for update ... 36, 52, 84, 88, 91-96, 102-104, 109, 112, 114, 146, 148-149, 205, 242
mask ... 52, 59, 65-67, 69, 87, 90, 92-93, 98, 116-117, 129, 131-132, 150, 219, 223, 225, 227-228, 231, 233, 245, 272

Mask For Use With File Access Modes ... **116**
maximum pathname length ... 45, 106, 243
MAX_CANON ... **45**, 105, 125-126
MAX_CHAR ... 215
MAX_INPUT ... **45**, 105, 125-126, 130, 215
may ... **23**
MB_LEN_MAX ... 44
medium ... 114, 155, 157-158, 161, 274, 277-279
memory management ... 39, 53, 173
metafile ... 167
MIN ... 126-127, 133, 262
Minimal Changes to Existing Application Code ... **175**
Minimal Changes to Historical Implementations ... **175**, 196
Minimal Interface, Minimally Defined ... **174**
mkdir ... **92**, 35, 44, 61, 90, 92, 96, 99, 102, 162, 180, 206, 241, 245-246
mkfifo ... **93**, 35, 44, 61, 90, 93, 99, 102, 196, 241, 245
mknod ... 196, 245
mktime ... **142**, 142-143, 150, 195
mode ... **31**
mode field ... 129, 131-132, 155, 157-158
modem access ... 28, 31, 87, 91, 101, 116-119, 124, 257
modem connection ... 132, 135
modem control lines ... 132, 135
Modem Disconnect ... **128**, 57, 128
modem line control ... 132, 135
modem lines ... 125-127, 132, 135, 181, 259
modem status lines ... 132
mode_t ... **40**, 87, 90, 92-93, 97-98, 101, 208, 245, 247
modf ... **141**
modification time ... 97, 103-104, 158, 160, 231, 249
modtime ... **104**, 249
mount ... 196, 201, 203, 207, 246-247, 249
mount point ... 201, 203, 207, 246, 249
mounted file system ... 196, 201, 247, 249
mtime ... 156, 158
Multi-volume archives ... 277
Multics ... 279
multiple groups option ... 25, 73
Multiple Volumes ... **162**, 277-278
Mumps ... 167
name field ... 131, 156-158, 160-161, 212, 276
name path ... 41, 91-92, 94-97, 101, 105-106, 157, 206, 210
name.h ... 29, 44, 74, 76, 80, 152, 161, 179, 211, 236-237, 239, 242, 277
namespace pollution ... 192, 208, 212-213
NAME_MAX ... **45**, 30, 36, 38, 45, 48, 53, 83, 85-86, 89, 91-95, 97, 100-106, 216, 249
National Body Conforming POSIX Application ... **25**
NBS ... 168, 272
nbyte ... 111-114, 252, 254
NCCS ... 129, 133
NDL ... 168
network connection ... 123, 132, 196, 263
networked systems ... 22, 167, 171, 180, 189, 196, 205, 247, 252, 257
networked transfers ... 252
Networking Standards ... **167**, 5, 167
NGROUPS_MAX ... **44**, 25, 34, 44, 73, 81
NL ... 126-130, 133
nlink_t ... **40**, 97, 208
No Super-User, No System Administration ... **174**
nodename ... **77**, 237
NOFLSH ... 132-133
nohup ... 219
non-canonical mode input processing ... 125-126, 133
Non-Canonical Mode Input Processing ... **126**
non-local jumps ... 141, 150, 183, 223, 272
NUL ... 142
NULL ... 27, 42, 51, 55, 65-66, 74, 77, 79-80, 84-86, 104, 144, 146, 152-153, 225, 233, 242-244, 268-269
null byte ... 30, 32, 42, 45, 51, 74, 79, 83, 86, 205, 277
null character ... 29-30, 32, 42, 51, 74, 79, 83, 86, 205, 241
null pathname ... 32, 36, 45, 80, 86, 161, 239, 277

null pointer ... 27, 42, 51, 74, 77, 79-80, 84-86, 104, 144, 146, 152-153, 268-269
null signal ... 57, 62-63, 65-66, 225, 232-233
null string ... **29**, 31, 42, 45, 51, 74, 79, 144, 241, 269
null-terminated character array ... 51, 77, 157
null-terminated character string ... 51, 157
null-terminated filename ... 51, 83
null-terminated pathname ... 80
null-terminated string ... 51, 80, 157
Numerical Limits ... **44**
object compatibility ... 174, 209, 251, 263
object file ... 29-30, 32, 193, 241, 257
octal ... 157-158, 160, 277
odd parity ... 131-132
off_t ... **40**, 97, 118, 120-121, 208, 252, 257-258
oflag ... 87, 89, 116, 244, 256
oflag Values For open ... **116**
open ... **87**
open file ... **31**, 29-31, 36-39, 44-45, 49, 51-52, 56, 84-85, 87-90, 94, 96, 100, 102, 105, 109-111, 113, 115-121, 123-124, 136, 138, 145-149, 188, 201, 219, 221, 239, 244, 247, 251, 255-257, 260, 269-271
open file description ... **31**, 29-31, 49, 51, 87, 109, 111, 117-118, 121, 136, 145-149, 201, 251, 255-256, 270-271
open file descriptor ... 30-31, 37, 49, 51, 56, 84-85, 87, 89, 100, 102, 105, 109-111, 113, 115, 117-121, 136, 146-148, 201, 247, 251, 255, 269
open flag bits ... 118
open function ... 36, 52, 84, 87-90, 100, 102, 109, 111, 113, 117-118, 121, 132, 136, 146-149, 174, 184, 201, 245, 247, 269-270
open instance ... 201, 264
Open Software Foundation ... 168
Open System Guidelines ... **166**
Open Systems ... 5, 163, 168
opendir ... **83**, 43, 83-85, 157, 161, 180, 242
Opening a Terminal Device File ... **123**
OPEN_MAX ... **45**, 38, 45, 81, 84, 109-110, 120, 184
operating environment ... 3, 21, 163, 171, 173, 181-182, 193
operating system documentation ... **24**, 3, 24, 173
OPOST ... 131
optional error ... 23, 207, 217, 233, 244, 250, 257
optional facilities ... 47-48
optional features ... 23, 187
optional_actions ... 136-137
OR ... 47, 54, 87, 99, 101, 129, 131-132
Organization of the Standard ... **183**
orphan ... 31, 57, 60, 124, 201-202, 222-223, 228, 261
orphaned process group ... **31**, 57, 60, 124, 201-202, 222-223, 228, 261
OSI Model ... 167
output baud rates ... 135, 264
output characters ... 127-128, 130-131
output control value ... 131
Output Modes ... **131**, 127, 131
output primitives ... 83, 109, 245, 250, 255
output processing ... 123-125, 127, 129-131, 133, 186
output queue ... 125, 130, 133
owner ... 12, 30, 35, 40, 52, 74, 88, 92-93, 97-99, 101-104, 157-161, 220, 248, 274
owner ID ... 30, 52, 92, 97, 99, 101-104, 158, 160, 220, 248
O_ACCMODE ... 116-117
O_APPEND ... 87, 113, 116
O_CREAT ... 87-90, 116
O_EXCL ... 87-89, 116
O_NDELAY ... 245, 250-251, 255-256
O_NOCTTY ... 88, 116, 124, 244
O_NONBLOCK ... 88-89, 109, 112-116, 123, 125, 127, 132, 175, 184, 245, 250-251, 253-256
O_RDONLY ... 87-88, 116, 245
O_RDWR ... 87-89, 116, 245
O_TRUNC ... 88-90, 116, 244
O_WRONLY ... 87-89, 116, 245
parameter ... 4, 124, 129, 136, 181-182, 192, 198, 224, 229, 233, 246, 249, 252, 259, 262
PARENB ... 131-132
parent directory ... **31**, 36, 49, 84, 88-89, 92-96, 241, 244

parent process ... **31**, 31-32, 34, 49-50, 52-53, 55-57, 59-60, 62, 71, 78, 84, 145, 198-199, 202, 216-217, 220, 222, 228-229, 232, 235, 237-238, 244, 269
parent process ID ... **31**, 34, 49-50, 52, 71, 198, 222, 235
parity ... 129-132, 157, 161, 263
parity error ... 130, 263
PARMRK ... 129-130
PARODD ... 131-132
Pascal ... 167, 191
passwd ... **152**
passwd database ... 152, 272
passwd file ... 201, 272
passwd Structure ... **152**
passwd.h ... 209
password ... 159, 257, 272
password database ... 257, 272
PATH ... **41**, 4, 41, 51, 53, 184, 210, 219
path prefix ... **32**, 36, 41, 51, 53, 89, 91-94, 96-97, 100-104, 106, 157
pathconf ... **105**, 36, 46, 48, 61, 105-106, 196, 216, 249-250
pathname ... **32**, 28-30, 32-33, 35-36, 38-39, 41, 44-46, 48, 51, 53, 79-80, 85-87, 89, 91-97, 100-106, 126, 157-158, 160-161, 194, 196, 203-206, 215-216, 218-219, 239-241, 243, 249-250, 256, 274-277
pathname component ... **32**, 30, 32, 36, 38-39, 48, 53, 85-86, 89, 91-95, 97, 100-104, 106, 216, 249
pathname resolution ... **36**, 28-30, 32-33, 35-36, 194, 203, 205, 241
Pathname Variable Values ... **45**
PATH_MAX ... **45**, 32, 38, 45, 53, 85-86, 89, 91-95, 97, 100-106, 215, 240, 243-244, 249, 276-277
pause ... **68**, 56, 61, 68-70, 227, 230, 233-235, 253
pclose ... 220-221
PCTS ... 168
pending signals ... 49, 52, 59-60, 62-63, 66-67, 69, 217, 226-228, 233
permission ... 4, 29-31, 34-35, 37, 39, 47, 53, 62-63, 85-87, 89-104, 106, 123, 155, 157-158, 160-161, 195, 220, 230, 232, 235, 244, 247-249, 261
permission bits ... 30, 35, 87, 90, 92-93, 98-99, 101, 157-158, 248
permission checking ... 100-101, 230, 232, 248
perror ... **141**, 149, 207, 271
pgrp_id ... 139-140
PHIGS ... 167
physical end ... 114
pid ... 54, 56, 62-63, 75-76, 209, 231-232
PID_MAX ... 209
pid_t ... **40**, 32-33, 40, 49, 54, 62, 71, 75, 118, 139, 209
pipe ... **109**, 30, 32, 40, 44-45, 57, 61, 94, 98-100, 106, 109-115, 121, 195, 198, 202, 228, 244, 251, 253-255, 269, 271
pipe definition ... **32**
pipeline ... 30, 32, 40, 57, 61, 94, 98-100, 106, 109-115, 121, 195, 198, 202, 228, 244, 251, 253-255, 269, 271
PIPE_BUF ... **45**, 105, 114, 253-255
PIPE_MAX ... 255
popen ... 221, 270
port ... 123, 129, 132, 168, 175, 191, 212-213, 227, 264
portability specifications ... 47-48, 173, 216, 260
portable application ... 42, 44-45, 172-173, 188-190, 192, 204-205, 209, 217, 220, 225-226, 229-231, 233, 239-240, 258-259, 263-264
portable filename character set ... **32**, 36, 41, 157, 161, 183, 202, 205, 210
portable library ... 225, 258
portable mechanism ... 223, 226, 229, 258, 261
Portable Operating System Interface ... 3, 21, 163
portable pathname ... 32
portable way ... 225-226, 244, 264
POSIX ... **3**, 5, 22-26, 42-44, 47, 166-168, 172, 174-175, 177-178, 180-188, 190-194, 196-198, 200-201, 204, 208, 210-215, 218, 223-225, 229, 234-236, 240-241, 245-247, 250-251, 253, 256, 258, 260, 265, 268-270, 274, 276-277
POSIX and the C Standard ... **180**
POSIX Extensions ... 23-24, 26, 175, 190, 214-215, 251
POSIX Symbols ... **42**, 42-43, 192, 208
posix.h ... 216
pow ... **141**

prefix ... 32, 36, 41, 51, 53, 89, 91-94, 96-97, 100-104, 106, 156-158, 193, 206, 213, 218, 276
prefix field ... 157-158, 276
Prime Meridian ... 143, 265
primitive system data types ... 40, 208, 247
Primitive System Data Types ... **40**
primitives ... 40, 49, 63, 83, 109, 165, 208, 216, 222, 227, 234, 245, 247, 250, 255
printf ... **141**, 149, 165-166, 172, 191, 207, 218, 225, 271, 281
privilege ... **32**, 28, 30, 32, 35, 40, 48, 72-73, 91, 94, 101-104, 106, 155, 158-159, 161-162, 165, 193, 203, 236, 244, 247-248, 274, 276
privileged operation ... 40, 236
process ... **32**
process creation ... 33, 49, 83, 87-88, 90, 92-93, 216-217, 221
Process Environment ... 71, 235
process group ... **32**, 9, 28-34, 40, 44, 48-49, 52, 54, 56-58, 60, 62-63, 71-73, 75-76, 87, 92-93, 99, 102-103, 113, 115, 123-124, 127-128, 130, 133, 136, 139-140, 155, 164, 171, 179-180, 197-203, 205, 216-217, 221-223, 227-228, 231, 236-237, 241, 248, 261, 264-265, 275
process group ID ... **32**, 29-34, 44, 48-49, 52, 62, 71-73, 75-76, 87, 92, 99, 102-103, 139, 198, 200, 203, 236-237, 241, 265
process group leader ... **33**, 75-76, 124, 202, 236
process group lifetime ... **33**, 29, 32-34
process ID ... **33**, 29-34, 40, 44-45, 48-50, 52, 54-57, 62, 71-76, 87, 92-93, 99, 101-104, 118-119, 139-140, 183, 198, 200, 203, 209, 219-220, 222, 231, 235-237, 241, 265
process identification ... 71, 235
process image ... 37, 39, 50-53, 148, 216, 219-220
process image file ... 50-53, 148, 219-220
process lifetime ... **33**, 29, 31-34, 81, 232, 240, 250
Process Primitives ... **49**, 216, 227
Process Termination ... **53**, 53-54, 184, 200, 220, 222, 226
Process Times ... 78, 238
processor scheduling delays ... 68
programming errors ... 37, 123, 189, 207, 225, 244, 253, 255, 278
prompt ... 173, 197-199, 211, 226
protection information ... 155
prototype ... 43, 192
PS1 ... 211
PS2 ... 211
ptrace ... 206, 221, 233
putc ... **141**, 149, 271
putchar ... **141**, 149, 271
putenv ... 239
puts ... **141**, 119, 130, 141, 149, 174, 261, 271
pwd.h ... **152**, 41, 44, 152, 159, 162, 209
pw_dir ... 152
pw_gid ... 152
pw_name ... 152
pw_shell ... 152
pw_uid ... 152
qsort ... **141**, 182
queue ... 44-45, 125-126, 130-131, 133, 215, 224, 261-262
queue_selector ... 137-138
QUIT character ... 127, 133
race conditions ... 198
radix ... 41
raise ... 60-61, 66, 179, 183
rand ... **141**, 6-7, 9, 141
range ... 25, 37, 77-78, 110, 143, 174, 186, 191, 196, 209, 214, 226, 251, 267, 275
Rationale and Notes ... **171**
raw mode ... 259
read ... **111**
read by group ... 58, 124, 157, 161, 199, 261
read by others ... 29, 112, 157, 161, 256
read by owner ... 157, 161
read function ... 36, 39, 84, 100, 109, 111-113, 121, 126, 137, 147-149, 197, 199, 207, 227, 230, 241, 247, 252-253, 255-256, 259, 261, 269
read operations ... 84, 113, 124, 126, 147-149, 242, 247, 251, 256-257, 278
read request ... 35, 37, 112, 118, 125, 127, 133, 148, 277
read-only file system ... **33**, 40, 89, 91, 93, 95-97, 101-104, 207-208

readahead ... 147, 271
readdir ... **83**, 43, 83-85, 180, 242-243
real group ID ... **33**, 31, 33, 52, 71-72, 100, 235
real time ... 33, 68-69, 78, 234-235, 238
real user ID ... **33**, 31, 33-34, 45, 52, 62, 71-72, 235
realloc ... **141**
realtime ... 5, 166, 224, 234, 238
Realtime Extensions ... **166**
Referenced C Language Routines ... **141**
region ... 118-120, 251, 256-257
REGTYPE ... 156
regular file ... **33**, 29-30, 33, 39, 50, 53, 88, 97-99, 102-103, 112-114, 118, 156, 158-159, 161-162, 202, 241, 244, 246, 248, 256, 273, 275-277
Related Standards ... **163**, 3, 22, 163-164, 172, 178, 182, 279
Related Standards and Documents ... **178**
Related Standards—Open System Environment ... **163**
relative pathname ... **33**
release ... **77**, 180, 188, 212, 223, 226, 232, 241, 245-246, 266, 272
reliable ... 38, 179, 183, 224, 246-247
reliable queueing ... 224
reliable signal ... 179, 183, 224
remove ... **141**
Remove Directory Entries ... 94, 246
remove function ... 27, 94-95, 111, 119, 149, 246, 272
rename ... **96**, 61, 91, 95-97, 141, 180, 182, 245-246
renaming directories ... 96, 246
renaming dot ... 246
reposition ... 120, 258
requested access ... 35, 91, 101, 118-119
Required Signals ... **57**
restore signal masks ... 67, 69, 150, 231, 272
return value ... 3-4, 36-37, 50, 52-53, 55-56, 63-81, 84-87, 89-95, 97, 100-106, 109-113, 115, 117, 119-121, 126-127, 134-140, 144-145, 149, 152-153, 181, 191, 203, 206-207, 217, 231, 234-235, 237-238, 240, 242, 247, 249-250, 252-253, 255, 258, 264, 268, 270
returned argument ... 54-55, 65, 73, 77, 86, 96, 117, 229, 234-235, 243, 245, 250, 252
returning zero ... 37, 50, 55-56, 63-65, 67, 70, 72-74, 76, 80, 84-86, 91-97, 100-104, 109, 111-113, 121, 126-127, 135-136, 138, 140, 187, 206, 233, 235, 242, 247, 250, 255
rewind ... **141**
rewinddir ... **83**, 43, 83-84, 180, 243
rewinding ... 141, 149, 190, 271
rewrite existing file ... 90, 245
rmdir ... **95**, 61, 94-95, 97, 180, 207, 245
root directory ... **33**, 36, 52, 95, 196, 203, 205-206, 246
root file system ... 196, 203
run-time ... 25, 44-45, 47, 81, 196, 214, 225, 240, 249-250
run-time facility ... 240, 249
Run-Time Increasable Values ... **44**, 25, 44
Run-Time Invariant Values (Possibly Indeterminate) ... **45**
run-time invariant values ... 45, 214, 250
run-time limits ... 196
running process ... 216
R_OK ... 47, 101
samefile ... 208
save ... 150, 183, 206, 219, 230, 233, 264, 268, 272, 274
saved process group ... 31, 33, 52, 227
saved process group ID ... 31, 33, 52
saved set-group-ID ... **33**, 31, 33, 47, 52, 72-73, 179, 235
saved set-user-ID ... **33**, 33-34, 47, 52, 62, 72-73, 179, 235
sa_flags ... 64-65, 233
sa_handler ... 64-65
sa_mask ... 64-65, 233
SA_NOCLDSTOP ... 60, 65, 198, 233
sbrk ... 182
scanf ... **141**, 149, 191, 271
SCHAR_MAX ... 44
SCHAR_MIN ... 44
scheduler ... 68, 165, 178, 231, 234-235, 238
scheduling ... 68-69, 127, 165-166, 178, 226, 234-235, 238
scheduling delays ... 68, 234-235
seconds since the Epoch ... **34**, 29, 34, 77, 99, 104, 142, 185, 194-195, 265

secure implementation ... 232, 247-248
security ... 35, 63, 166, 168, 193, 195, 200, 203, 220, 231-232, 237, 248, 261, 265, 272, 274-275
security label ... 232
seekdir ... 174, 243
seeking ... 26, 40, 112-113, 120, 147-149, 190, 270-271
SEEK_CUR ... 47, 119, 121
SEEK_END ... 47, 119, 121, 258
SEEK_SET ... 47, 119, 121, 271
select ... 30, 157, 161, 189, 198, 253, 280
semaphores ... 166, 230, 256
session ... **34**, 28, 30-31, 34, 52, 57, 62, 75-76, 124, 140, 167, 198, 201-202, 222, 231, 236-237, 265
session leader ... **34**, 28, 34, 75-76, 124, 202
session lifetime ... **34**
session process group leader ... 75-76, 124, 202
Set File Access and Modification Times ... 103, 249
Set File Creation Mask ... 90, 245
Set Foreground Process Group ID ... **139**
set gid ... 157, 161
set group ID ... 34, 52, 72, 75, 87, 92-93, 99, 102, 139, 179, 200, 235-236, 241, 265
set process group ... 34, 52, 62, 72, 75, 92, 99, 123-124, 130, 139, 200, 217, 236, 241, 265
set uid ... 157-158, 161, 274
set user ... 52, 72, 87, 92-93, 99, 102, 104, 158, 162, 179, 197, 202, 235, 250, 258, 274
set-group-ID ... 31, 33, 47, 52, 72-73, 103, 179, 219-220, 235
set-user-ID ... 33-34, 47, 52, 62, 72-73, 103, 179, 219-220, 235
setbuf ... **141**
setgid ... **72**, 29, 33, 61, 72-73, 248, 257
setgrent ... 272
setgroups ... 236
sethostid ... 238
sethostname ... 238
setjmp ... **141**, 150, 183, 235, 272
setjmp ... **150**, 141, 150, 183, 235, 272
setjmp.h ... 44, 150
setlocale ... **144**, 141, 144-145, 267-269
setpgid ... **75**, 33-34, 61, 75-76, 139, 187, 198-199, 236-237
setpwent ... 272
setsid ... **75**, 33-34, 61, 63, 75-76, 124, 139, 236
setuid ... **72**, 29, 33, 52-53, 61, 72-73, 230-231
setvbuf ... 270
Seventh Edition ... 4, 178
shall ... **23**
shell ... 5, 22, 123, 155, 164-165, 185, 197-201, 210-211, 217-218, 222, 225, 228, 231, 236-237, 261
Shell and Utilities ... **164**
should ... **23**
SHRT_MAX ... 44
SHRT_MIN ... 44
SIGABRT ... **57**, 53, 57, 224
sigaction ... **64**, 44, 57, 59-61, 63-68, 70, 75, 121, 150, 223, 225, 227, 231, 233, 265, 272
sigaddset ... **63**, 44, 61, 63-64
SIGALRM ... **57**, 68-69, 234-235
SIGBUS ... 225
SIGCHLD ... **58**, 57-58, 60-61, 65, 198, 225-226, 228, 233
SIGCLD ... 224-225, 228, 233
SIGCONT ... **58**, 57-59, 62, 198, 222-223, 225, 227-228, 231
sigdelset ... **63**, 44, 61, 63-64
sigemptyset ... **63**, 44, 61, 63-64, 232
SIGEMT ... 225
sigfillset ... **63**, 44, 61, 63-64, 232
SIGFPE ... **57**, 60, 66, 225, 227
SIGHUP ... **57**, 128, 222-223, 228
SIGILL ... **57**, 60, 66, 225
SIGINT ... **57**, 38, 57, 127, 130, 200, 217, 224
SIGIOT ... 224-225
sigismember ... **63**, 44, 61, 63-64
SIGKILL ... **57**, 60-61, 65-66, 224, 226-228, 230, 233
siglongjmp ... **150**, 44, 69, 150, 183, 207, 230, 272
SIGMA Project ... 178
signal ... **34**
Signal Actions ... **60**
Signal Concepts ... **57**
Signal Effects on Other Functions ... **61**
Signal Generation and Delivery ... **59**
signal handler ... 38, 65, 118, 198, 206, 224, 229-230, 270

signal handling ... 57, 65, 68, 200, 226, 270
signal interfaces ... 28, 174, 223, 231
signal mask ... 52, 59, 65-67, 69, 150, 219, 223, 225, 227-228, 231, 233, 272
signal names ... 57, 124, 174, 179, 200, 221, 224-225, 232
Signal Names ... **57**
signal-catching function ... 54, 60-62, 64-65, 67-69, 223-224, 227-231, 233-235
signal.h ... **57**, 34, 38, 44, 49, 52-53, 55-57, 59, 62-68, 89, 111, 120, 136, 150, 203, 220, 226-228, 233-234, 261
Signals ... **57**
sigpending ... **67**, 44, 52-53, 61, 64, 67-68, 223-224, 232
SIGPIPE ... **57**, 115, 207
sigprocmask ... **66**, 44, 52-53, 59, 61, 64-68, 150, 226, 231, 233-235, 272
SIGQUIT ... **57**, 38, 57, 127
sigreturn ... 223
SIGSEGV ... **57**, 60-61, 66, 225
sigsetjmp ... **150**, 44, 69, 150, 183, 272
sigsetops ... **63**, 57, 63, 66-68
sigset_t ... 57, 63-64, 66-67, 224
sigstack ... 223
SIGSTOP ... **58**, 58-61, 65-66, 225, 228, 233
sigsuspend ... **67**, 44, 59, 61, 64-68, 150, 227, 230, 233-235, 272
SIGSYS ... 225
SIGTERM ... **57**, 224
SIGTRAP ... 225
SIGTSTP ... **58**, 58-60, 128, 200, 225, 228-229
SIGTTIN ... **58**, 58-60, 113, 124, 199-200, 225, 228, 261
SIGTTOU ... **58**, 58-60, 115, 124, 132-133, 136, 198, 200, 225, 228, 261
SIGUSR1 ... **57**, 224-225
SIGUSR2 ... **57**, 224-225
sigvec ... 233
SIG_BLOCK ... 66
SIG_DFL ... **57**, 52, 57, 60, 64, 201, 224, 226-228
SIG_IGN ... **57**, 52, 57, 60, 64, 198, 200, 219, 226-228
SIG_SETMASK ... 66
SIG_UNBLOCK ... 66
simple abnormal termination ... 220
sin ... **141**
single-byte functions ... 128
sinh ... **141**
size field ... 97, 158-159, 232, 275-277
slash ... **34**, 30, 32-36, 41, 51, 85, 157, 205-206, 274-275
sleep ... **69**, 61, 68-70, 119, 221, 229-230, 234-235, 257
socket ... 250, 269
Solely by POSIX ... **182**
Solely by X3J11 ... **182**
solidus ... 34
source code level ... 21-22, 164, 263
source form ... 240
source level ... 3, 21-22, 155, 164, 251, 263
Source, Not Object, Portability ... **174**
Special Characters ... **127**, 57-58, 123, 125-128, 133-134
Special Control Characters ... **133**, 127-128, 133
special files ... 28-30, 32, 34, 39, 88, 92-93, 98, 111-112, 123, 128, 130, 134, 156, 159, 161, 179, 193, 195, 201, 245, 248, 258, 270, 276
special functions ... 22, 28, 32, 39, 127-128, 133-134, 191, 213, 231, 241, 245, 248, 270
Specific Derivations ... **179**
specific implementation ... 24, 26, 37, 44-46, 54, 59, 63, 100, 106-107, 136, 144, 146, 158, 188-189, 193, 196, 202-203, 214, 218, 221, 225-227, 231, 233, 237, 248-249, 259, 264, 266
specific interfaces ... 23, 173, 175, 193, 196, 231, 264
speed_t ... 134-135
sprintf ... **141**
SQL ... 165, 168
sqrt ... **141**
srand ... **141**
sscanf ... **141**
stacked alarm request ... 68
Standard C ... 174, 178, 181, 185, 190-191, 211, 239, 258, 270-271
standard input ... 123, 145, 253, 264, 271, 275
standard output ... 123, 145, 161,

263-264
Standards Closely Related to the 1003.1 Document ... **164**
START ... 128, 130-131, 133, 138
start/stop input control ... 129-130
start/stop output control ... 129-130
stat ... **99**
stat file types ... 97-98
stat function ... 27, 33, 37, 49, 61, 84, 97, 99-100, 158, 191, 195, 206, 213, 227, 229, 247, 264
stat structure ... 97-98, 100, 191, 212, 229, 247, 264
stat Structure ... **97**
stat.h ... **97**
static data ... 74, 79-80, 152-153
stat_loc ... 54-56, 221
stderr ... 145, 149
STDERR_FILENO ... 145
stdin ... 145, 270
STDIN_FILENO ... 145
stdio ... 269-270
stdio.h ... 44, 74, 79, 145-146
stdlib.h ... 79, 192-193
stdout ... 145, 270
STDOUT_FILENO ... 145
stop signal ... 55, 58-61, 65, 124, 133, 198-200, 202, 227-228, 233
stopped children ... 65, 200, 222-223, 233
stopped process ... 31, 54, 57, 59-61, 124, 133, 197-200, 202, 221-223, 227-228, 233, 261
strcat ... **141**
strchr ... **141**, 141-142
strcmp ... **141**
strcpy ... **141**
strcspn ... **142**
stream ... 49, 56, 84-85, 133, 137, 145-149, 160, 162, 179, 182-183, 193, 207, 221, 242-243, 253, 269-271, 274
streams inter-process communications ... 179
strftime ... **142**, 142-143, 150
strictly conforming implementation ... 22-23, 44, 47, 188, 218, 229
Strictly Conforming POSIX Application ... **25**, 22, 24-25, 44, 47, 188, 190, 218, 234-235
string ... 29, 31-32, 38-42, 45, 51, 53, 74, 79-80, 86, 89, 91-95, 97, 100-104, 106, 141, 144-146, 157, 160, 205, 211, 237, 239-241, 266-269
String Handling ... 141, 267
string terminator ... 237, 241
strlen ... **142**, 241
strncat ... **141**
strncmp ... **141**
strncpy ... **141**
strpbrk ... **142**
strrchr ... **142**
strspn ... **142**
strstr ... **142**
strtok ... **142**
structure ... 21, 24, 28, 33, 36, 43, 47, 57, 64-65, 77-78, 83-84, 97-98, 100, 104, 118, 129, 135-137, 142, 151-152, 156, 161, 165, 172, 184, 188, 191, 202, 204, 207-209, 212, 217, 224, 229-230, 232-233, 237, 241-242, 247, 249, 257, 259, 262, 264
structure elements ... 257
structure members ... 43, 64, 77-78, 97, 104, 129, 151-152, 208, 212, 237, 249
st_atime ... **97**, 52, 84, 88, 92-93, 97, 99, 109, 112, 146, 148-149, 242, 249, 269
st_ctime ... **97**, 88, 91-97, 99, 102-104, 109, 114, 148-149, 269
st_dev ... **97**, 247
st_gid ... **97**
st_ino ... **97**, 247
st_mode ... **97**, 97-98
st_mtime ... **97**, 88, 91-97, 99, 109, 114, 148-149, 249, 269
st_nlink ... **97**, 208
st_size ... **97**
st_uid ... **97**
subscript ... 133-134
subshell ... 199
suffix ... 157
summer time ... 142-143, 265-267
super-user ... 174, 193, 203, 231, 235, 245-248, 274
supplementary group ID ... **34**, 30-31, 34, 44, 48, 52, 72-73, 102-103, 203, 236, 248
supported ... **23**
supported languages ... 26-27, 43, 173, 265, 269
SUSP ... 127-128, 133

suspend ... 31, 53-54, 67-69, 128, 130, 132, 138, 147, 197, 200, 216, 230, 234, 237
suspend character ... 130, 197, 200
suspend process execution ... 31, 53-54, 68-69, 197, 216, 230, 234
suspended input ... 128
suspended output ... 128, 130, 138
SVID ... 178-179, 208, 217, 241, 243, 246, 272, 280
symbolic constant ... 4, 25, 31, 47-48, 57, 74, 79-80, 101, 105, 121, 123, 136, 138, 156, 172, 174, 184-185, 189, 216, 237, 239-240, 247
Symbolic Constants ... **47**
Symbolic Constants for the access Function ... **47**
Symbolic Constants for the access() Function ... **47**
Symbolic Constants for the lseek Function ... **47**
symbolic link ... 203, 273, 276-277
symbolic name ... 4, 37, 44, 57, 74, 80, 105, 174, 194, 207, 225, 238, 277
Symbols From The C Standard ... **42**
SYMTYPE ... 156, 158
sys/dir.h ... 241
sys/stat.h ... **97**, 30, 36, 44, 53, 62, 87, 89-90, 92-94, 97-104, 159, 162, 184, 208-209, 244, 247
sys/times.h ... 44, 78
sys/types.h ... **40**, 49, 54, 62, 71-73, 75, 83, 87, 90, 92-93, 99, 101-103, 117, 120, 139, 208, 211-212, 238, 247
sys/utsname.h ... 44, 76-77
sys/wait.h ... **54**, 44, 54-55
sysconf ... **80**, 45, 47, 61, 80-81, 196, 225, 240, 249-250
sysname ... **77**
system ... **34**
system administration ... 5, 166, 174, 188, 193, 239, 246
System Administration Extensions ... **166**
system CPU time ... 78
system databases ... 22, 151, 196, 201, 272
system documentation ... 3-4, 24, 51, 77, 173, 179
system identification ... 76, 237, 257
System III ... 4, 178-180, 202, 212, 223, 237, 248, 250, 253, 258-259, 280
System III/V ... 250
System Interface ... **164**
system process ... **34**, 22, 31-34, 37, 49-50, 56-57, 59, 61-63, 68-69, 78, 83, 87, 94, 97, 119, 125, 165, 179, 181, 189, 193, 196, 200, 214, 222, 225-227, 230-231, 236, 238, 240, 246, 249-250, 257
system services ... 21-22, 49, 83, 173, 190-191, 194, 196, 216
System V ... 4, 178-180, 194, 200-201, 212, 215, 219, 222-226, 228, 230-231, 233-237, 241, 245-246, 248-251, 253, 255-259, 265, 272, 276, 280-281
S_IRGRP ... 98-99
S_IROTH ... 99
S_IRUSR ... 98-99
S_IRWXG ... 98-99
S_IRWXO ... 98-99
S_IRWXU ... 98-99
S_ISBLK ... 98
S_ISCHR ... 98
S_ISDIR ... 98, 184
S_ISFIFO ... 98
S_ISGID ... 99, 101-103, 247-248
S_ISREG ... 98
S_ISUID ... 62, 99, 101, 103, 247-248
S_IWGRP ... 98-99
S_IWOTH ... 99
S_IWUSR ... 98-99
S_IXGRP ... 98-99
S_IXOTH ... 99
S_IXUSR ... 98-99
tabs ... 262-263
tan ... **141**
tanh ... **141**
tape ... 39, 155-156, 253, 259, 273-274, 278-279
tape mark ... 278
tapecntl ... 259
tar ... **155**, 155-156, 175, 180, 273-276
tar Header Block ... **156**
tar.h ... 156
tc ... 259-260
tcdrain ... **137**, 44, 61, 137-138
tcflow ... **137**, 44, 61, 137-138
tcflush ... **137**, 44, 61, 137-138
tcgetattr ... **136**, 44, 48, 61, 134, 136-137, 199, 264

tcgetpgrp ... **139**, 61, 123, 139, 199, 265
TCIFLUSH ... 138
TCIOFF ... 138
TCIOFLUSH ... 138
TCION ... 138
TCOFLUSH ... 138
TCOOFF ... 138
TCOON ... 138
TCSADRAIN ... 136
TCSAFLUSH ... 136
TCSANOW ... 136
tcsendbreak ... **137**, 44, 61, 137-138
tcsetattr ... **136**, 44, 48, 61, 134-137, 199, 260, 264
tcsetpgrp ... **139**, 61, 76, 123, 139-140, 198-199, 237, 265
teletypewriter ... 207
telldir ... 174, 243
TERM ... **41**
termcntl ... 259
terminal ... **34**
Terminal Access Control ... **124**, 30, 58, 123-124, 200, 229, 261, 264
terminal device ... 28, 30, 34, 80, 88, 123-125, 127-131, 134-135, 138, 215, 230, 239, 244, 256, 259-260, 262-264
terminal device file ... 28, 30, 123-125, 127-129, 134, 244, 256, 260, 262
terminal device name ... 80, 239
terminal driver ... 197-198, 200, 232, 261
terminal file ... 28, 30, 34, 36, 80, 105, 112, 123-125, 127-130, 134, 136-140, 193, 199, 201, 239, 244, 256, 260-262
Terminal Identification ... 239
terminal input ... 28, 44-45, 123-125, 129-132, 135, 138, 215, 261, 263
terminal instead of terminal device ... 239
terminal interface ... 28, 31, 34, 123, 128, 135-136, 197, 207, 258-260, 263-264
terminal parameters ... 124, 136
Terminate Process ... 56, 200, 221
terminated child ... 54-55, 58, 61, 78, 216, 222, 233
terminated children ... 56-57, 78, 220, 222, 225, 228
termination ... 33, 52-57, 59, 149, 184, 200, 219-222, 225-226, 229, 232, 270
termination consequences ... 57, 222
termination signal ... 53, 55, 57, 220, 222, 225-226, 229, 232, 270
Terminology ... **23**, 4, 21, 23, 186
termios ... **123**
termios Baud Rate Values ... **135**
termios c_cc Special Control Characters ... **133**
termios c_cflag Field ... **131**
termios c_iflag Field ... **129**
termios c_lflag Field ... **132**
termios information ... 259
termios Structure ... **129**
termios structure ... **129**, 135-137, 259, 262, 264
termios.h ... **129**, 44, 48, 129, 131-133, 135-138
termios_p ... 135-136
test suite ... 168, 188
Text vs. binary file modes ... 181, 269
TGEXEC ... 157
TGREAD ... 157
TGWRITE ... 157
The C Language and X3J11 ... **174**
The Controlling Terminal ... **124**
TIME ... 126-127, 133, 262
time ... **77**
time remaining ... 33, 59, 68
time standard ... 6, 12, 36, 48, 78-79, 143, 164, 176, 194-195, 219, 230-231, 238, 247, 251, 257, 259, 263, 265-266
time zone ... 41, 142-143, 150, 265-267, 272
time-accounting information ... 78
time-related fields ... 36, 99-100
time.h ... 29, 39-40, 44-45, 68, 77-78, 80, 104, 137, 142-143, 194-195, 197, 207, 211, 214, 223, 230, 235, 238-239, 251, 262-263, 265-266
timeout facility ... 257
timer operations ... 84, 149, 234
timer value ... 34, 36, 70, 77-78, 104, 109, 126, 133, 142-143, 150, 158, 182, 194-195, 205, 235, 237-238
times ... **78**
times function ... 36, 41, 52, 59, 68-70, 77-79, 84, 102, 104, 109, 126, 142, 148-150, 158, 173, 210, 227, 230-231, 235, 238-239, 247, 259, 265,

269
times.h ... 78, 103, 238, 240, 249, 269
time_t ... **40**, 42, 77-78, 97, 104, 195, 211, 238
timing windows ... 217
TIOCGPGRP ... 265
TIOCSPGRP ... 265
title ... 272
tloc ... 77
TMAGIC ... 156, 159
TMAGLEN ... 156
tmpfile ... **141**, 149, 271
tmpnam ... **141**
tms_cstime ... **78**, 49, 52, 78
tms_cutime ... **78**, 49, 52, 78
tms_stime ... **78**, 49, 52, 78
tms_utime ... **78**, 49, 52, 78
toascii ... 182
TOEXEC ... 157
Token Bus ... 167
Token Ring ... 167
tolower ... **141**
Topical Index ... **186**
TOREAD ... 157
TOSTOP ... 115, 124, 132-133, 198
toupper ... **141**
TOWRITE ... 157
toy implementation ... 214
traditional function ... 221, 229-232, 260
traditional implementations ... 218-219, 221, 224, 227, 229-232, 234-235, 246, 261
trailer ... 161, 278
trailing null ... 32, 158
translate ... 22, 128, 130, 155, 174, 176, 214, 265
translation ... 181, 188, 263, 268
Translation vs. Execution Environment ... 188
transmission ... 127, 130-131, 137-138
transmitting data ... 130-131, 137-138
transportable archive ... 155, 160
transportable medium ... 274
trojan horse ... 193
trusted system ... 5, 166, 168, 248
Trusted System Extensions ... **166**
Trusted Systems ... **168**
TSGID ... 157
TSUID ... 157
TSVTX ... 157-158
tty ... 207, 215
ttyname ... **80**, 182, 239
TUEXEC ... 157
TUREAD ... 157
TUWRITE ... 157
TVERSION ... 156
TVERSLEN ... 156
type arguments ... 181, 192, 247, 258-259, 269
typeflag ... 156-158, 276
types.h ... **40**, 98, 129, 208-209, 252, 257
Typographical Conventions ... **184**
TZ ... **41**, 142-143, 150, 210, 265-267
tzname ... 143, 150
tzset ... **150**, 44, 143, 150
UCHAR_MAX ... 44
UID ... 157-158, 193, 274
UID_MAX ... 209
uid_t ... **40**, 34, 40, 71-72, 97, 102, 152, 208-209
UINT_MAX ... 44, 234-235
ulong ... 208
ULONG_MAX ... 44
umask ... **90**, 44, 52-53, 61, 87, 89-90, 92-94, 245
umount ... 201
uname ... **76**, 44, 61, 76-77, 156-157, 159, 180, 237
uname Structure Members ... **77**
undefined ... **23**, 23-24, 38, 60, 66, 87, 105, 156, 186-187, 191, 206, 221, 229, 239, 270, 277
undefined results ... 23, 66, 87, 156, 187, 270
undefined term ... 187, 239
underlying function ... 136, 147-149, 182, 270
ungetc ... **141**
unique ... 5, 30, 32, 37, 42, 49, 99, 116, 133, 196, 202, 224, 247
unistd.h ... **47**, 24-25, 42-43, 47-48, 80-81, 101, 105-106, 121, 145, 159, 185, 188, 213, 216, 225, 239
United States ... 266
units ... 125, 159, 191, 198-199, 215, 266
UNIX ... 3-4, 171-175, 178-179, 185, 190, 192-194, 200-201, 203, 205, 213, 265
unlink ... **94**, 61, 91, 94-97, 99, 111, 149, 207-208, 245-246
unlock ... 111, 116, 119-120, 257
unmount ... 246, 250

unpredictable behavior ... 143
unrecoverable error ... 253, 255
unsigned ... 68-69, 111, 113, 129, 135, 158, 208-209, 213, 234-235, 245, 252, 257-258
unsigned argument ... 234-235, 245
unsigned char ... 111, 113, 209
unsigned offsets ... 258
unsigned short ... 208
unspecified ... **24**, 59, 61, 65, 69, 73, 83, 186-187, 191, 205, 222, 233, 238, 241, 259
uppercase ... 42, 185, 204
US Government Standards ... **168**
USASCII ... 204
USENIX ... 9, 176, 178, 281
User CPU time ... 78
user database ... **151**, 41, 74, 151-152, 257, 272-273
user ID ... **34**, 29-31, 33-34, 40, 45, 52, 62, 71-72, 74, 87, 92-93, 97, 99-100, 102, 151-152, 179, 203, 208, 220, 230-231, 235-236, 248, 275
User Identification ... 180, 235
User Name ... 74, 236
user processes ... 29-31, 33-34, 44-45, 50, 52, 62, 71-74, 87, 92-93, 99, 102, 104, 147, 155, 193, 197, 201, 203, 214, 220, 228, 231, 235, 244, 247, 250, 257
user utility ... 157-159, 161-162, 274, 276
ushort ... 208
ushort_t ... 208
USHRT_MAX ... 44
USTAR ... 275
ustat ... 61, 246
utilities ... 5, 123, 141, 155, 157, 161, 164-165, 222, 227, 273-275, 278
utility ... 4-5, 155-159, 161-162, 164, 180, 182, 218, 260, 270, 274, 276-279
utimbuf ... 104
utimbuf structure ... 104
utime ... **103**, 44, 61, 99, 103-104, 249
utime.h ... 44, 104
utsname.h ... **76**
valid file descriptor ... 80, 100, 107, 110-111, 113, 115, 120-121, 137-140, 247, 251
variable errno ... 37, 81, 86, 106, 182, 186, 206, 230, 250
variable number ... 27, 37, 181, 229, 256
variable parameter lists ... 229
VDM ... 167
vector ... 151
VEOF ... 133, 263
VEOL ... 133, 263
VERASE ... 133
verb ... 251
verification suites ... 188, 216
Verification Testing ... **166**, 3, 5, 166, 216
version ... **77**
Version 7 ... 174, 178-180, 205, 212, 223, 230, 234, 237-238, 241, 248, 251, 253, 258-259, 275, 279-280
vfprintf ... 149, 271
vhangup ... 201, 222
VINTR ... 133
virtual time ... 234
visibility of symbols ... 42-43
VKILL ... 133
VMIN ... 133-134, 263
void ... 56-57, 60, 64, 83, 150, 221, 229, 242-243
vprintf ... 149, 271
VQUIT ... 133
VSTART ... 133-134
VSTOP ... 133-134
VSUSP ... 133
VTIME ... 133-134, 263
wait ... **54**
Wait for Process Termination ... 54, 200, 220
wait function ... 33, 38, 54-56, 78, 126, 132, 137, 186, 216, 220-221, 227, 229-230, 233, 261
wait.h ... **54**
wait3 ... **220**
waitpid ... **54**, 33, 38, 44, 53-56, 61, 78, 198, 202, 209, 216, 220-221, 229, 232-233
wall clock time ... 234
WeirdNIX ... **177**
WERASE ... 262
WEXITSTATUS ... **55**
WIFEXITED ... **55**
WIFSIGNALED ... **55**
WIFSTOPPED ... **55**, 221
WNOHANG ... **54**, 56, 233
WORD_BIT ... 213
working directory ... **35**, 29, 35-36, 41,

52, 85-86, 95, 106, 151-152, 194, 210, 220, 243, 246, 249
Working Directory Pathname ... 86, 243
working documents ... 165-166, 172, 175, 178, 180, 273, 275
Working Documents ... **180**
Working Group ... **175**, 3-4, 6, 9, 123, 155, 164-166, 171-178, 180-183, 185, 189, 195, 205-207, 209, 214, 216, 224, 226-227, 229, 231, 233-235, 237-238, 240, 242-243, 246-248, 251-252, 254-255, 258-261, 272-273, 275
write ... **113**
write by group ... 58, 124, 133, 156-157, 161, 261, 275
write by others ... 113-114, 157, 161, 211, 256
write by owner ... 157, 161
write function ... 36, 39, 100, 104, 109, 113-115, 147-149, 182, 197-198, 207, 219, 227, 229, 234-235, 252-253, 255-256, 258-259, 261, 269
Write requests ... 114
Writing Data and Output Processing ... **127**
WSTOPSIG ... **55**
WTERMSIG ... **55**
WUNTRACED ... **54**, 54-55, 198, 220-221
W_OK ... 47, 101
X.212 ... 167
X.25 ... 167
X.400 ... 167
X/Open ... 9, 176, 178, 239, 260
X/OPEN Portability Guide ... 168, 178, 212
X3.159-198x ... 26, 28, 164, 174, 178, 185
X3H3.6 ... 167
X3J11 ... 164, 174, 178, 180-183, 185, 192, 210, 212, 221-222, 252
X3J11 Rationale ... 178, 180-183, 185
X_OK ... 47, 101, 248
yardstick ... 3
zero-filled ... 157
zero-valued bits ... 130, 137
_asm_builtin_atoi ... 193
_BSD ... 208, 212
_exit ... **56**, 53, 55-57, 61, 75, 128, 147-148, 183, 221-223, 229, 270
_longjmp ... 272
_PC_CHOWN_RESTRICTED ... 105
_PC_LINK_MAX ... 105
_PC_MAX_CANON ... 105
_PC_MAX_INPUT ... 105
_PC_NAME_MAX ... 105
_PC_NO_TRUNC ... 105, 216
_PC_PATH_MAX ... 105
_PC_PIPE_BUF ... 105
_PC_VDISABLE ... 105
_POSIX_ARG_MAX ... **44**, 44-45
_POSIX_CHILD_MAX ... **44**, 44-45
_POSIX_CHOWN_RESTRICTED ... **48**, 25, 48, 102-103, 105, 249
_POSIX_JOB_CONTROL ... **47**, 25, 31, 47, 76, 81, 123, 139
_POSIX_LINK_MAX ... **44**, 44-45
_POSIX_MAX_CANON ... **44**, 44-45
_POSIX_MAX_INPUT ... **44**, 44-45
_POSIX_NAME_MAX ... **44**, 44-45
_POSIX_NGROUPS_MAX ... **44**
_POSIX_NO_TRUNC ... **48**, 36, 39, 48, 53, 85-86, 89, 91-95, 97, 100-106, 216
_POSIX_OPEN_MAX ... **44**, 44-45
_POSIX_PATH_MAX ... **44**, 44-45
_POSIX_PIPE_BUF ... **44**, 44-45
_POSIX_SAVED_IDS ... **47**, 52, 62, 72-73, 81, 219
_POSIX_SOURCE ... **43**, 212
_POSIX_VDISABLE ... **48**, 105, 128, 134, 184
_POSIX_VERSION ... **47**, 81, 237
_SC_ARG_MAX ... 81
_SC_CHILD_MAX ... 81
_SC_CLK_TCK ... 81, 240
_SC_NGROUPS_MAX ... 81
_SC_OPEN_MAX ... 81
_SC_SAVED_IDS ... 81
_SC_VERSION ... 81
_setjmp ... 272
_SYSIII ... 208, 212
_SYSV ... 212
_tolower ... 182
_toupper ... 182
_USR_GROUP ... 212

Acknowledgements

We wish to thank the following organizations for donating significant computer, printing, and editing resources to the production of this standard: /usr/group, Amdahl Corporation, Digital Equipment Corporation, MASSCOMP, and UniSoft Corporation.

Also we wish to thank the organizations employing the members of the Working Group and the Balloting Group for both covering the expenses related to attending and participating in meetings, and donating the time required both in and out of meetings for this effort.

/usr/group*
/usr/group/cdn
Absolut Software
ACM
ADDAMAX
Aeon Technologies, Inc.
AFUU
AGS Information Services
Air Force Institute of Technology
Aktiengesellschaft
Alcyon Corporation
Alliant Computer Systems
Alsup
Amdahl Corporation*
Analysts International
ANFOR
Anistics, Inc.
ANSI
Apollo Computer, Inc.
Apple Computer
Applicon
Applied Network Technology
APT Data Services
Archives Ltd.
Associated Computer Experts
AT Computing/Toernooiveld
AT&T*
AT&T Bell Laboratories
AT&T Information Systems
AT&T UNIX Europe Ltd.
Atlanta UNIX Users Group
ATTAGE
Australian Government Dept. of Science
Australian UNIX Systems User Group
Automation Resource Group
Automation Technologies
Baldwin Information Processing
Barrister Information Systems Corporation
Batelle Columbus Labs
BBN Communications Corporation
BBN Laboratories
Bell Communications Research
Bell-Northern Research Ltd.
Billy Shakespeare & Company
Boeing Aerospace Company
BP America Research & Development
Brake Systems, Inc.
British Airways
British Olivetti Ltd.
British Standards Institute
British Telecom
Brown & Sharpe
Bull Sems
Bull, Inc.
Burroughs Corporation
Burtek, Inc.
C. S. Draper Lab, Inc.
California State University
California University
Calspan Corporation
Carnegie-Mellon University
CCTA, Riverwalk House
Celerity Computing
Central Computer & Telecommunications
Central Intelligence Agency
Centre National D'Etudes des Telecomm.
CFI
Charles River Data Systems*
Chorus Systems
Citicorp Transaction Tech., Inc.
Classic Conferences
Cleveland State University
CMC Ltd.
CommUNIXations
COMPASS
Compugraphic Corporation
Computer Design
Computer Systems Engineering
Computer Systems News
Computer Systems Resources, Inc.
Computer Task Group
Computer Works
Computer X, Inc.
Computerworld
Comtek
Concord Data Systems
Concurrent Computer Corporation*
Control Data Corporation*
Convergent Technologies
Convex Computer Corporation
COSMIC
Cray Research, Inc.
Cullinet Software, Inc.
Custom Development Environments
Dana Computers
Dansk Data Elektronik A/S DDE

Dansk Standardiseringsradd
Dartmouth College
Data Connection
Data General Corporation
Data Logic Ltd.
Data Systems Engineering
DataBoard, Inc.
Datamation
Datamension Corporation
Datapoint Corporation
DEC International
Defence Communication Agency
Deutsches Institut fur Normung
DGM&S, Inc.
Digital Equipment Corporation*
Digital Equipment GmbH
Digital Sound Corporation
Directorate Land Armament
Dravo Automation Sciences (DAS)
Eastman-Stuart Ltd.
Eclipse Systems Corporation
EDS Corporation
EDV Zentrum der TU Wien
Electronic Engineering Times
Electrospace Systems, Inc.
Emerald City
Emerging Technologies Group, Inc.
ENEA DATA Svenska AB
Epicom
Epilogue Technology Corporation
Ericsson Information Systems AB
ETA Systems, Inc.
European Laboratory for Particle Physics
European UNIX
Eurotherm International
EUUG*
Exxon Chemical Pakistan Ltd.
Fern Universitat
FGAN/FFM
Fidcom System Ltd.
Flexible Automation
Flexible Computer Corporation
Floating Point Systems
Ford Aerospace
Ford Motor Company
Fortune Systems Corporation
Free Software Foundation, Inc.
Fujitsu America, Inc.
Future Tech, Inc.
FUUG (Finland)
Gartner Group
Geac Computer International
GEC Telecomms Ltd.
General Dynamics
General Electric Company
General Motors Corporation
General Motors Technical Center
George Mason University
Georgia Institute of Technology
Gilbert International
Gould CSD
Gould Electronics
Gould SEL*
Government Computer News
Grebyn Corporation
Grumman Aircraft
Grumman Data
Grupo de Redes de Computadores
GUUG (West Germany)
Harrie Corporation
Harris Corporation Computer Systems Division
Hayden Publishing
HCR Corporation
Hewlett-Packard Company*
Hi-Tech Editorial, Inc.
High Tech Publishing Company, Inc.
Honeywell
Honeywell ISI
Honeywell Ltd.
Hughes Aircraft Company
Human Computing Resources
IAI MBT
IBM Corporation*
IBM Federal Systems Division
IBM Research Center
IBM Thomas J. Watson Research Center
ICL
IDC
IEEE Computer Magazine
IEEE Computer Society
IEEE Micro
IEEE Reflector
IEEE Standards Office
Imperial College
Industrial Technology Institute
InfoPro Systems
Information Concepts Pty Ltd.
Ing. C. Olivetti & C., SPA
Inside Information
Institute for Defense Analysis
Institute of Software Academia Sinica
Instruction Set
International Bureau of Software Test
Integrated Systems Design, Inc.
Intel Corporation
INTERACTIVE Systems
Intergraph Corporation
International Computers Ltd.
Internet Systems
Iowa State University
Irish UNIX Systems Users Group
Iskra Automatika
ISO-OSCRAC
Israel Aircraft Industries
Italian UNIX Systems User Group i2u
Itom International
ITT
ITUSA
Japanese Industrial Stds. Committee
Johnson Controls, Inc.
K.E.T.R.I.
KAIST

Kendall Square Research Corporation
Key Tech
King Abdulziz University
Korean UNIX User Group
Lachman Associates, Inc.
Lawrence Livermore National Laboratory
Liberty Mutual Research Center
Lisp Machine, Inc.
LM Ericsson
Lockheed
Lockheed S.O.C
Logicon, Inc.
Lowell University
LSI Appl. Info. & Learning Center
LTV Aerospace & Defense
M.B.F. Systems, Inc.
Maharishi International University
Mallinckrodt Institute of Radiology
Mark Williams Company
Markor
Martin Marietta Aerospace
Martin Marietta Data Systems
MASSCOMP
Maxim Technologies, Inc.
McDonnell Douglas
McDonnell Douglas Computer Company
McGill University
Mercury Computer Systems, Inc.
Meridian Software Systems
Microfocus
Micrology, Inc.
Microsoft Corporation
Microtex
Mindcraft, Inc.
Mini-Micro Systems
MIPS Computer Systems
MIT-LCS
Mitre Corporation
Modcomp
Modular Systems
Molecular Genetics, Inc.
Monarch Data Systems
Mortice Kern Systems, Inc.
Motorola Israel, Inc.
Motorola, Inc.
MT XINU
NAPS, Inc.
NASA
NASA Ames Research Center
NASA Johnson Space Center
NASA Kennedy Space Center
National Bureau of Standards
National Cancer Institute
National Computer Security Center
National Electrical Manufacturers Assoc.
National Research Council
National UNIX Systems User Group
Naval Ocean Systems Center
Naval Postgraduate School
Naval Surface Weapons Center
Naval Underwater Systems Center
NBI, Inc.
NCR*
NCR Cambridge
NEC Info Systems
Nederlands Normalisatie-instituut
New Media Development Center
New Zealand UNIX Systems User Group, Inc.
Nicolet Instrument Corporation
Nikkei Byte, Nikkei McGraw-Hill, Inc.
Nippon Tel & Tel Corporation
NIUIS 77
Nixdorf Computer AG
Norsk Data Ltd.
North Holland P&C
Northern N.E. UNIX User Group
Northwestern Bell
Northwestern University
Novell
Novon Research Group
NRAO
OCLC
Ohio State University
Olivetti
Olivetti & C., SPA
Oren Yuen Associates
Osterreichesches Normungsinstitut
Oxford Systems, Inc.
Pacific Marine Technology
Palladium Data Systems
Patricia Seybold's Office Computing Grp.
Perennial
Philadelphia Area Computer Society
Philips and Picker Medical Systems
Philips Data Systems
Phillips Publishing, Inc.
Planning Research Corporation
Plum Hall, Inc.
Plus Five Computer Services
Polish Computer Society
Politecnic Di Torino-DIP Automatic
Politecnico Di Mikeno
Portland Community College
Prime Computer, Inc.
Princeton University
Programming Concepts, Inc.
Purdue University
Pyramid Technology
Pyramid Technology Ltd.
Rabbit Software Corporation
RCA Automated Systems
RDA Logicon
Relcom, Inc.
Release 1.0
Richmond Computerware
Ricoh Systems, Inc.
RJO Enterprises
Rogers State University
Rolm Corporation
Rolm Mil-Spec Computers
S. J. Lipton, Inc.
SAH Consulting

Sandia National Labs
Santa Cruz Operation
SAS Institute, Inc.
Schlumberger Well Services
SCI Systems, Inc.
Scientific Computer Systems
SCS
SDRC
Seattle/UNIX Group
Seay Systems, Inc.
SEI Information Technology
SELENIA SP. A
Sequent Computer Systems
Shreerang Society
Siemens AG
Sigma, IPA
Silicon Valley Net
Simpact Associates, Inc.
Singapore UNIX Association
Softech, Inc.
Softtech
Software Engineering Company
Software Laboratories Ltd.
Software News
Software Productivity Consortium
Software Research Associates, Inc.
Software-PEI
SoHar
Spectra-Tek U.K. Ltd.
Sperry Corporation
Sperry Ltd. Education Centre
Sphinx Ltd.
SSC
St. Lawrence College
Standards Council of Canada
Statskontoret
Stellar Computer, Inc.
Stewart Research Enterprises
Stratus Computer
Strong Consulting
Structural Dynamics Research Corporation
Structured Methods
Summit Computer Systems, Inc.
Sun Microsystems, Inc.*
Syntactics
Syntek Systems
System House, Inc.*
Systems & Software Magazine
Systems Development Corporation
Tampere University of Technology
Tandem Computers, Inc.
Technical Solutions, Inc.
Technical University of Delft
Technische Universitat Berlin
Teknekron Infoswitch
Tektronix, Inc.*
Telephone Organization of Thailand
Teli Foretagssystem
Tenis Software Consulting, Inc.
Texas Instruments, Inc.
Texas Instruments-DSG
Texas Internet Consulting
The C Journal, InfoPro Systems
The Charles Stark Draper Laboratory, Inc.
The Foxboro Company
The Wollongong Group
Thinking Machines Corporation
TIS
Torch Computers Ltd.
Toshiba Corporation
Treasury Board of Canada
TRW
Tsinghua University Computer Dept.
U.S. Air Force
U.S. Army
U.S. Army Ballistic Research Lab.
U.S. Army ISEC
U.S. D.O.T. Systems Center
U.S. Department of Defense
U.S. Department of H.U.D.
U.S. Dept. of Commerce NOAA/NOS
U.S. Federal Judicial Center
U.S. West Advanced Technologies
U.K. UNIX Systems User Group
UNI Karlsruhe, Informatik II
Uni'C' Computer Systems
Uni-Ops
Unigram:X
Unigroup of New York, Inc.
UNIQ Digital Technologies
UniSoft Corporation
UniSoft Ltd.
UNISYS*
UNIT-C
Universitat Dortmund
Universitat Zurich-Irchel
University of California, Berkeley
University of California, Irvine
University of California, Los Angeles
University of Colorado
University of Copenhagen
University of Evansville
University of Hong Kong
University of Indonesia
University of Lowell
University of Maryland
University of Minnesota, Dept. of CS
University of Nevada, Las Vegas*
University of New Mexico
University of Portland
University of Santa Cruz
University of South Florida
University of Surrey
University of Technology
University of Tennessee
University of Texas at Arlington
University of Texas at Austin
University of Toronto
University of Utah
University of Victoria
University of Vienna Dept. of Statistics
University of Wisconsin-Milwaukee

University of Zagreb
UNIX Houston
UNIX Review
UNIX Systems
UNIX Technologies
UNIX User Group Austria
UNIX Users of Minnesota
UNIX World
UNIX-C Club
UNIX/WORLD
USENIX Association*
USSR State Committee for Standards
Varian
Venturcom
Verdix Corporation
Veritas Technology, Inc.
Videoton
Virginia Polytech & State University
Wang Laboratories, Inc.
West Virginia University
Western Digital
Whitesmiths, Ltd.
Wind River Systems, Inc.
Woods Hole Oceanographic Institution
World UNIX & C
X/Open
XIOS Systems Corporation
Yates Ventures

In the preceding list, the organizations marked with an asterisk (*) have hosted 1003 Working Group meetings since the group's inception in 1985, providing useful logistical support for the ongoing work of the committees.

For Applications Portability Look to IEEE's POSIX Standard.

IEEE Std 1003.1-1988 defines a standard operating system interface and environment based on the UNIX* Operating System documentation to provide for the portability of applications software at the source-code level, between computer systems from multiple vendors.

In its present form, the standard focuses primarily on the C Language interface to the operating system. It complements related standards for computer languages, database management, and computer graphics, and is intended to be used by both application developers and system implementors.

The IEEE POSIX specification provides a solid base for systems procurement and evaluation. Specifying POSIX conformance will allow system purchasers to productively manage their software environments and to achieve the benefits from applications portability. Hardware and software manufacturers will have a clear specification to follow in designing systems and applications for the open software environment.

This standard also constitutes a major step in the industry to provide a comprehensive applications environment. The IEEE's POSIX committees are continuing POSIX-related standards work in areas such as POSIX-based Open System architecture (IEEE Project 1003.0), shell and utilities (IEEE Project 1003.2), test methods (IEEE Project 1003.3), real time systems interfaces (IEEE Project 1003.4), Ada language binding for POSIX (IEEE Project 1003.5), and systems security (IEEE Project 1003.6).

To be placed on our mailing list for these yet-to-be-published standards, please send your name, address, and the project numbers that interest you to: The Institute of Electrical and Electronics Engineers, Inc., The Standards Department, 445 Hoes Lane, PO Box 1331, Piscataway, NJ 08855-1331, USA.

**UNIX is a registered trademark of AT&T.*

ISBN 1-55937-003-