

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

- A comprehensive applications environment
 - Applications portability at the source code level
 - An applications interface to I/O, file system access and process management facilities

- Software Applications Developers
- System Designers/Engineers
- Hardware and Software Purchasers
- End users operating a UNIX OS environment

ISO/IEC JTC1/SC22/WG2

ISBN 1-55937-061-0

Library of Congress Catalog Number 90-084554

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

© Copyright 1990 by

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

*No part of this publication may be reproduced in any form,
in an electronic retrieval system or otherwise,
without the prior written permission of the publisher.*

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

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

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

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

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

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

IEEE Standards documents are adopted by the Institute of Electrical and Electronics Engineers without regard to whether their adoption may involve patents on articles, materials, or processes. Such adoption does not assume any liability to any patent owner, nor does it assume any obligation whatever to parties adopting the standards documents.

Contents

	PAGE
Foreword	viii
Introduction	ix
Section 1: General	1
1.1 Scope	1
1.2 Normative References	2
1.3 Conformance	2
Section 2: Terminology and General Requirements	9
2.1 Conventions	9
2.2 Definitions	10
2.3 General Concepts	21
2.4 Error Numbers	23
2.5 Primitive System Data Types	27
2.6 Environment Description	27
2.7 C Language Definitions	29
2.8 Numerical Limits	34
2.9 Symbolic Constants	37
Section 3: Process Primitives	41
3.1 Process Creation and Execution	41
3.1.1 Process Creation	41
3.1.2 Execute a File	42
3.2 Process Termination	46
3.2.1 Wait for Process Termination	47
3.2.2 Terminate a Process	49
3.3 Signals	51
3.3.1 Signal Concepts	51
3.3.2 Send a Signal to a Process	56
3.3.3 Manipulate Signal Sets	57
3.3.4 Examine and Change Signal Action	58
3.3.5 Examine and Change Blocked Signals	60
3.3.6 Examine Pending Signals	62
3.3.7 Wait for a Signal	62
3.4 Timer Operations	63
3.4.1 Schedule Alarm	63
3.4.2 Suspend Process Execution	64
3.4.3 Delay Process Execution	65
Section 4: Process Environment	67
4.1 Process Identification	67
4.1.1 Get Process and Parent Process IDs	67

	PAGE
4.2 User Identification	68
4.2.1 Get Real User, Effective User, Real Group, and Effective Group IDs	68
4.2.2 Set User and Group IDs	68
4.2.3 Get Supplementary Group IDs	70
4.2.4 Get User Name	71
4.3 Process Groups	72
4.3.1 Get Process Group ID	72
4.3.2 Create Session and Set Process Group ID	72
4.3.3 Set Process Group ID for Job Control	73
4.4 System Identification	74
4.4.1 Get System Name	74
4.5 Time	75
4.5.1 Get System Time	75
4.5.2 Get Process Times	76
4.6 Environment Variables	77
4.6.1 Environment Access	77
4.7 Terminal Identification	78
4.7.1 Generate Terminal Pathname	78
4.7.2 Determine Terminal Device Name	79
4.8 Configurable System Variables	80
4.8.1 Get Configurable System Variables	80
Section 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	86
5.2.1 Change Current Working Directory	86
5.2.2 Get Working Directory Pathname	87
5.3 General File Creation	88
5.3.1 Open a File	88
5.3.2 Create a New File or Rewrite an Existing One	91
5.3.3 Set File Creation Mask	91
5.3.4 Link to a File	92
5.4 Special File Creation	94
5.4.1 Make a Directory	94
5.4.2 Make a FIFO Special File	95
5.5 File Removal	96
5.5.1 Remove Directory Entries	96
5.5.2 Remove a Directory	98
5.5.3 Rename a File	99
5.6 File Characteristics	101
5.6.1 File Characteristics: Header and Data Structure	101
5.6.2 Get File Status	103
5.6.3 Check File Accessibility	104
5.6.4 Change File Modes	106
5.6.5 Change Owner and Group of a File	107

	PAGE
5.6.6 Set File Access and Modification Times	108
5.7 Configurable Pathname Variables	110
5.7.1 Get Configurable Pathname Variables	110
Section 6: Input and Output Primitives	113
6.1 Pipes	113
6.1.1 Create an Inter-Process Channel	113
6.2 File Descriptor Manipulation	114
6.2.1 Duplicate an Open File Descriptor	114
6.3 File Descriptor Deassignment	115
6.3.1 Close a File	115
6.4 Input and Output	116
6.4.1 Read from a File	116
6.4.2 Write to a File	118
6.5 Control Operations on Files	121
6.5.1 Data Definitions for File Control Operations	121
6.5.2 File Control	121
6.5.3 Reposition Read/Write File Offset	127
Section 7: Device- and Class-Specific Functions	129
7.1 General Terminal Interface	129
7.1.1 Interface Characteristics	129
7.1.1.1 Opening a Terminal Device File	129
7.1.1.2 Process Groups	129
7.1.1.3 The Controlling Terminal	130
7.1.1.4 Terminal Access Control	130
7.1.1.5 Input Processing and Reading Data	131
7.1.1.6 Canonical Mode Input Processing	132
7.1.1.7 Noncanonical Mode Input Processing	132
7.1.1.8 Writing Data and Output Processing	133
7.1.1.9 Special Characters	133
7.1.1.10 Modem Disconnect	135
7.1.1.11 Closing a Terminal Device File	135
7.1.2 Parameters That Can Be Set	135
7.1.2.1 <i>termios</i> Structure	135
7.1.2.2 Input Modes	136
7.1.2.3 Output Modes	137
7.1.2.4 Control Modes	138
7.1.2.5 Local Modes	139
7.1.2.6 Special Control Characters	140
7.1.2.7 Baud Rate Values	141
7.1.3 Baud Rate Functions	141
7.1.3.1 Synopsis	141
7.1.3.2 Description	142
7.1.3.3 Returns	142
7.1.3.4 Errors	142
7.1.3.5 Cross-References	142
7.2 General Terminal Interface Control Functions	143
7.2.1 Get and Set State	143

	PAGE
7.2.2 Line Control Functions	145
7.2.3 Get Foreground Process Group ID	147
7.2.4 Set Foreground Process Group ID	148
Section 8: Language-Specific Services for the C Programming	
Language	151
8.1 Referenced C Language Routines	151
8.1.1 Extensions to Time Functions	152
8.1.2 Extensions to <i>setlocale()</i> Function	154
8.2 C Language Input/Output Functions	155
8.2.1 Map a Stream Pointer to a File Descriptor	156
8.2.2 Open a Stream on a File Descriptor	157
8.2.3 Interactions of Other <i>FILE</i> -Type C Functions	158
8.2.4 Operations on Files — the <i>remove()</i> Function	162
8.3 Other C Language Functions	162
8.3.1 Nonlocal Jumps	162
8.3.2 Set Time Zone	162
Section 9: System Databases	165
9.1 System Databases	165
9.2 Database Access	166
9.2.1 Group Database Access	166
9.2.2 User Database Access	167
Section 10: Data Interchange Format	169
10.1 Archive/Interchange File Format	169
10.1.1 Extended <i>tar</i> Format	169
10.1.2 Extended <i>cpio</i> Format	173
10.1.3 Multiple Volumes	177
Annex A (informative) Bibliography	179
A.1 Related Open Systems Standards	179
A.2 Other Standards	181
A.3 Historical Documentation and Introductory Texts	182
Annex B (informative) Rationale and Notes	185
B.1 Scope and Normative References	185
B.2 Definitions and General Requirements	196
B.3 Process Primitives	226
B.4 Process Environment	246
B.5 Files and Directories	253
B.6 Input and Output Primitives	264
B.7 Device- and Class-Specific Functions	273
B.8 Language-Specific Services for the C Programming	
Language	283
B.9 System Databases	293
B.10 Data Interchange Format	294
Annex C (informative) Header Contents Samples	301

	PAGE
Annex D (informative) Profiles	313
D.1 Definitions	313
D.2 Options in This Part of ISO/IEC 9945	314
D.3 Related Standards	315
D.4 Related Activities	315
D.5 Relationship to IEEE Draft Project 1003.0	315
Annex E (informative) Sample National Profile	317
E.1 (Example) Profile for Denmark	318
Identifier Index	321
Alphabetic Topical Index	327

TABLES

Table 2-1 – Primitive System Data Types	27
Table 2-2 – Reserved Header Symbols	31
Table 2-3 – Minimum Values	35
Table 2-4 – Run-Time Increasable Values	35
Table 2-5 – Run-Time Invariant Values (Possibly Indeterminate)	36
Table 2-6 – Pathname Variable Values	36
Table 2-7 – Invariant Value	37
Table 2-8 – Symbolic Constants for the <i>access()</i> Function	38
Table 2-9 – Symbolic Constants for the <i>lseek()</i> Function	38
Table 2-10 – Compile-Time Symbolic Constants	38
Table 2-11 – Execution-Time Symbolic Constants	39
Table 3-1 – Required Signals	52
Table 3-2 – Job Control Signals	52
Table 4-1 – <i>uname()</i> Structure Members	75
Table 4-2 – Configurable System Variables	80

Table 5-1	– stat Structure	101
Table 5-2	– Configurable Pathname Variables	111
Table 6-1	– cmd Values for fcntl()	122
Table 6-2	– File Descriptor Flags Used for fcntl()	122
Table 6-3	– l_type Values for Record Locking With fcntl()	122
Table 6-4	– oflag Values for open()	122
Table 6-5	– File Status Flags Used for open() and fcntl()	122
Table 6-6	– File Access Modes Used for open() and fcntl()	123
Table 6-7	– Mask for Use With File Access Modes	123
Table 6-8	– flock Structure	125
Table 6-9	– fcntl() Return Values	126
Table 7-1	– termios Structure	136
Table 7-2	– termios c_iflag Field	136
Table 7-3	– termios c_cflag Field	138
Table 7-4	– termios c_lflag Field	139
Table 7-5	– termios c_cc Special Control Characters	140
Table 7-6	– termios Baud Rate Values	141
Table 9-1	– group Structure	166
Table 9-2	– passwd Structure	167
Table 10-1	– tar Header Block	170
Table 10-2	– Byte-Oriented cpio Archive Entry	174
Table 10-3	– Values for cpio c_mode Field	175
Table B-1	– Suggested Feature Test Macros	222

Foreword

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

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for approval before their acceptance as International Standards. They are approved in accordance with procedures requiring at least 75% approval by the national bodies voting.

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

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

— *Part 1: System application program interface (API) [C language]*

— *Part 2: Shell and utilities* (under development)

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

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



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

Introduction

(This Introduction is not a normative part of ISO/IEC 9945-1 Information technology—Portable operating system interface (POSIX)—Part 1: System application programming interface (API) [C Language], but is included for information only.)

The purpose of this part of ISO/IEC 9945 is to define a standard operating system interface and environment based on the UNIX¹⁾ Operating System documentation to support application portability at the source level. This is intended for systems implementors and applications software developers.

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

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

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

Organization of This Part of ISO/IEC 9945

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

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

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

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

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

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

In publishing this part of ISO/IEC 9945, its developers simply intend to provide a yardstick against which various operating system implementations can be measured for conformance. It is *not* the intent of the developers to measure or rate any products, to reward or sanction any vendors of products for conformance or lack of conformance to this part of ISO/IEC 9945, or to attempt to enforce this part of ISO/IEC 9945 by these or any other means. The responsibility for determining the degree of conformance or lack thereof with this part of ISO/IEC 9945 rests solely with the individual who is evaluating the product claiming to be in conformance with this part of ISO/IEC 9945.

Base Documents

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

Background

The developers of POSIX.1 represent a cross-section of hardware manufacturers, vendors of operating systems and other software development tools, software designers, consultants, academics, authors, applications programmers, and others. In the course of their deliberations, the developers reviewed related American and international standards, both published and in progress.

Conceptually, POSIX.1 describes a set of fundamental services needed for the efficient construction of application programs. Access to these services has been

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

provided by defining an interface, using the C programming language, that establishes standard semantics and syntax. Since this interface enables application writers to write portable applications—it was developed with that goal in mind—it has been designated POSIX,⁴⁾ an acronym for Portable Operating System Interface.

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

Audience

The intended audience for ISO/IEC 9945 is all persons concerned with an industry-wide standard operating system based on the UNIX system. This includes at least four groups of people:

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

Purpose

Several principles guided the development of this part of ISO/IEC 9945:

Application Oriented

The basic goal was to promote portability of application programs across UNIX system environments by developing a clear, consistent, and unambiguous standard for the interface specification of a portable operating system based on the UNIX system documentation. This part of ISO/IEC 9945 codifies the common, existing definition of the UNIX system. There was no attempt to define a new system interface.

Interface, Not Implementation

This part of ISO/IEC 9945 defines an interface, not an implementation. No distinction is made between library functions and system calls: both are referred to as functions. No details of the implementation of any function are given (although historical practice is sometimes indicated in Annex B). Symbolic names are given for constants (such as signals and error numbers) rather than numbers.

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

115 **Source, Not Object, Portability**

116 This part of ISO/IEC 9945 has been written so that a program written
117 and translated for execution on one conforming implementation may
118 also be translated for execution on another conforming implementation.
119 This part of ISO/IEC 9945 does not guarantee that executable (object or
120 binary) code will execute under a different conforming implementation
121 than that for which it was translated, even if the underlying hardware
122 is identical. However, few impediments were placed in the way of
123 binary compatibility, and some remarks on this are found in Annex B.
124 See B.1.3.1.1 and B.4.8.

125 **The C Language**

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

128 **No Super-User, No System Administration**

129 There was no intention to specify all aspects of an operating system.
130 System administration facilities and functions are excluded from
131 POSIX.1, and functions usable only by the super-user have not been
132 included. Annex B notes several such instances. Still, an implementa-
133 tion of the standard interface may also implement features not in this
134 part of ISO/IEC 9945; see 1.3.1.1. This part of ISO/IEC 9945 is also not
135 concerned with hardware constraints or system maintenance.

136 **Minimal Interface, Minimally Defined**

137 In keeping with the historical design principles of the UNIX system,
138 POSIX.1 is as minimal as possible. For example, it usually specifies only
139 one set of functions to implement a capability. Exceptions were made in
140 some cases where long tradition and many existing applications
141 included certain functions, such as *creat()*. In such cases, as throughout
142 POSIX.1, redundant definitions were avoided: *creat()* is defined as a spe-
143 cial case of *open()*. Redundant functions or implementations with less
144 tradition were excluded.

145 **Broadly Implementable**

146 The developers of POSIX.1 endeavored to make all specified functions
147 implementable across a wide range of existing and potential systems,
148 including:

- 149 (1) All of the current major systems that are ultimately derived from
150 the original UNIX system code (Version 7 or later)
- 151 (2) Compatible systems that are not derived from the original UNIX
152 system code

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

- 155 (3) Emulations hosted on entirely different operating systems
156 (4) Networked systems
157 (5) Distributed systems
158 (6) Systems running on a broad range of hardware

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

161 **Minimal Changes to Historical Implementations**

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

- 170 (1) By standardizing an interface like one in an historical implemen-
171 tation; e.g., directories, or;
172 (2) By specifying an interface that is readily implementable in terms
173 of, and backwards compatible with, historical implementations,
174 such as the extended tar format in 10.1.1, or;
175 (3) By specifying an interface that, when added to an historical imple-
176 mentation, will not conflict with it, like B.6.

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

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

181 It should be noted that implementations will have different kinds of
182 extensions. Some will reflect "historical usage" and will be preserved for
183 execution of pre-existing applications. These functions should be con-
184 sidered "obsolescent" and the standard functions used for new applica-
185 tions. Some extensions will represent functions beyond the scope of
186 POSIX.1. These need to be used with careful management to be able to
187 adapt to future POSIX.1 extensions and/or port to implementations that
188 provide these services in a different manner.

189 **Minimal Changes to Existing Application Code**

190 A goal of POSIX.1 was to minimize additional work for the developers of
191 applications. However, because every known historical implementation
192 will have to change at least slightly to conform, some applications will
193 have to change. Annex B points out the major places where POSIX.1
194 implies such changes.

195 **Related Standards Activities**

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

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

- 200 (1) Language-independent service descriptions of this part of ISO/IEC 9945
- 201 (2) C, Ada, and FORTRAN Language bindings to (1)
- 202 (3) Shell and Utility facilities
- 203 (4) Verification testing methods
- 204 (5) Realtime facilities
- 205 (6) Secure/Trusted System considerations
- 206 (7) Network interface facilities
- 207 (8) System Administration
- 208 (9) Graphical User Interfaces
- 209 (10) Profiles describing application- or user-specific combinations of Open Sys-
210 tems standards for: supercomputing, multiprocessor, and batch exten-
211 sions; transaction processing; realtime systems; and multiuser systems
212 based on historical models
- 213 (11) An overall guide to POSIX-based or related Open Systems standards and
214 profiles

215 Extensions are approved as “amendments” or “revisions” to this document, follow-
216 ing the IEEE and ISO/IEC Procedures.

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

219 If you have interest in participating in the TCOS working groups addressing these
220 issues, please send your name, address, and phone number to the Secretary, IEEE
221 Standards Board, Institute of Electrical and Electronics Engineers, Inc., P.O. Box
222 1331, 445 Hoes Lane, Piscataway, NJ 08855-1331, and ask to have this forwarded
223 to the chairperson of the appropriate TCOS working group. If you have interest in
224 participating in this work at the international level, contact your ISO/IEC national
225 body.

6) A *Standards Status Report* that lists all current IEEE Computer Society standards projects is available from the IEEE Computer Society, 1730 Massachusetts Avenue NW, Washington, DC 20036-1903; Telephone: +1 202 371-0101; FAX: +1 202 728-9614. Working drafts of POSIX standards under development are also available from this office.

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

**Technical Committee on Operating Systems
and Application Environments (TCOS)**

Chair: Luis-Felipe Cabrera

Standards Subcommittee for TCOS

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

1003.1 Working Group Officials

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

Working Group

Steve Bartels
Robert Bismuth
James Bohem
Kathy Bohrer
Keith Bostic
Jonathan Brown
Tim Carter
Myles Connors
Landon Curt Noll
Dave Decot
Mark Doran
Glenn Fowler

Greg Goddard
Andrew Griffith
Rand Hoven
Randall Howard
Mike Karels
Jeff Kimmel
David Korn
Bob Lenk
Shane McCarron
John Meyer
Martha Nalebuff

Neguine Navab
Paul Rabin
Seth Rosenthal
Lorne Schachter
Steve Schwarm
Paul Shaughnessy
Steve Sommars
Ravi Tavakley
Jeff Tofano
David Willcox
John Wu

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

David Chinn	<i>Open Software Foundation Institutional Representative</i>	
Michael Lambert	<i>X/Open Institutional Representative</i>	
Heinz Lycklama	<i>UniForum Institutional Representative</i>	
Shane McCarron	<i>UNIX International Institutional Representative</i>	
Helene Armitage	William Henderson	Martha Nalebuff
David Athersych	Lee A. Hollaar	Barry Needham
Timothy Baker	Terrence Holm	Alan F. Nugent
Geoff Baldwin	Randall Howard	Jim Oldroyd
Steven E. Barber	Irene Hu	Craig Partridge
Robert Barned	Andrew Huber	John Peace
John Barr	Richard Hughes-Rowlands	John C. Penney
James Bohem	Judith Hurwitz	P. Plauger
Kathryn Bohrer	Jim Isaak	Gerald Powell
Robert Borochoff	Dan Iuster	Scott E. Preece
Keith Bostic	Richard James	Joseph Ramus
James P. Bound	Hal Jespersen	Wendy Rauch
Joseph Boykin	Michael J. Karels	Carol Raye
Kevin Brady	Sol Kavy	Wayne B. Reed
Phyllis Eve Bregman	Lorraine C. Kevra	Christopher J. Riddick
Fred Lee Brown, Jr.	Jeffrey S. Kimmel	Andrew K. Roach
A. Winsor Brown	M. J. Kirk	Robert Sarr
Luis-Felipe Cabrera	Dale Kirkland	Lorne H. Schachter
Nicholas A. Camillone	John T. Kline	Norman Schneidewind
Clyde Camp	Kenneth Klingman	Stephen Schwarm
John Carson	Joshua Knight	Richard Scott
Steven Carter	Andrew R. Knipp	Leonard Seagren
Jerry Cashin	David Korn	Glen Seeds
Kilnam Chon	Don Kretsch	Karen Sheaffer
Anthony Cincotta	Takahiko Kuki	Charles Smith
Mark Colburn	Thomas Kwan	Steven Sommars
Donald W. Cragun	Robin B. Lake	Douglas H. Steves
Ana Maria DeAlvare	Mark Lamonds	James Tanner
Dave Decot	Doris Lebovits	Ravi Tavakley
Steven Deller	Maggie Lee	Marc Teitelbaum
Terence Dowling	Greger Leijonhufvud	Donn S. Terry
Stephen A. Dum	Robert Lenk	Gary F. Tom
John D. Earls	David Lennert	Andrew Twigger
Ron Elliott	Donald Lewine	Mark-Rene Uchida
David Emery	Kevin Lewis	L. David Umbaugh
Philip H. Enslow	F. C. Lim	Michael W. Vannier
Ken Faubel	James Lonjers	David John Wallace
Kester Fong	Warren E. Loper	Stephen Walli
Kenneth R. Gibb	Roger Martin	Larry Wehr
Michel Gien	Martin J. McGowan	Bruce Weiner
Gregory W. Goddard	Marshall McKusick	Robert Weissensee
Dave Grindeland	Robert McWhirter	P. J. Weyman
Judy Guist	Paul Merry	Andrew Wheeler, Jr.
James Hall	Doug Michels	David Willcox
Charles Hammons	Gary W. Miller	Randall F. Wright
Allen Hankinson	James Moe	Oren Yuen
Steve Head	James W. Moore	Jason Zions
Barry Hedquist		

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

Marco W. Migliaro, *Chairman*

James M. Daly, *Vice Chairman*

Andrew G. Salem, *Secretary*

Dennis Bodson
Paul L. Borrill
Fletcher J. Buckley
Allen L. Clapp
Stephen R. Dillon
Donald C. Fleckenstein
Jay Forster*
Thomas L. Hannan

Kenneth D. Hendrix
John W. Horch
Joseph L. Koepfinger*
Irving Kolodny
Michael A. Lawler
Donald J. Loughry
John E. May, Jr.

Lawrence V. McCall
L. Bruce McClung
Donald T. Michael*
Stig Nilsson
Roy T. Oishi
Gary S. Robinson
Terrance R. Whittemore
Donald W. Zipse

*Member Emeritus

THIS PAGE WAS
BLANK IN THE ORIGINAL

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

Section 1: General

1.1 Scope

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

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

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

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

- (1) User interface (shell) and associated commands
- (2) Networking protocols and system call interfaces to those protocols
- (3) Graphics interfaces
- (4) Database management system interfaces
- (5) Record I/O considerations

- 17 (6) Object or binary code portability
- 18 (7) System configuration and resource availability
- 19 (8) The behavior of system services on systems supporting concurrency
- 20 within a single process

21 This part of ISO/IEC 9945 describes the external characteristics and facilities that
22 are of importance to applications developers, rather than the internal construc-
23 tion techniques employed to achieve these capabilities. Special emphasis is
24 placed on those functions and facilities that are needed in a wide variety of com-
25 mercial applications.

26 This part of ISO/IEC 9945 has been defined exclusively at the source-code level.
27 The objective is that a Strictly Conforming POSIX.1 Application source program
28 can be translated to execute on a conforming implementation.

29 1.2 Normative References

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

- 37 (1) ISO/IEC 646: 1983,¹⁾ *Information processing—ISO 7-bit coded character set*
38 *for information interchange.*
- 39 (2) ISO/IEC 9899: ...,²⁾ *Information technology—Programming languages—C.*

40 1.3 Conformance

41 1.3.1 Implementation Conformance

42 1.3.1.1 Requirements

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

44 1) Under revision. (This notation is meant to explicitly reference the 1990 Draft International
45 Standard version of ISO/IEC 646.)

46 ISO/IEC documents can be obtained from the ISO office, 1, rue de Varembé, Case Postale 56, CH-
47 1211, Genève 20, Switzerland/Suisse.

48 2) To be approved and published.

- 49 (1) The system shall support all required interfaces defined within this part
50 of ISO/IEC 9945. These interfaces shall support the functional behavior
51 described herein.
- 52 (2) The system may provide additional functions or facilities not required by
53 this part of ISO/IEC 9945. Nonstandard extensions should be identified
54 as such in the system documentation. Nonstandard extensions, when
55 used, may change the behavior of functions or facilities defined by this
56 part of ISO/IEC 9945. The conformance document shall define an
57 environment in which an application can be run with the behavior
58 specified by the standard. In no case shall such an environment require
59 modification of a Strictly Conforming POSIX.1 Application.

60 1.3.1.2 Documentation

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

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

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

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

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

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

89 The system documentation should also contain the information found in the con-
90 formance document.

91 1.3.1.3 Conforming Implementation Options

92 The following symbolic constants, described in the subclauses indicated, reflect
93 implementation options for this part of ISO/IEC 9945 that could warrant require-
94 ment by Conforming POSIX.1 Applications, or in specifications of conforming sys-
95 tems, or both:

- | | | |
|----|---------------------------|---|
| 96 | {NGROUPS_MAX} | Multiple groups option (in 2.8.3) |
| 97 | {_POSIX_JOB_CONTROL} | Job control option (in 2.9.3) |
| 98 | {_POSIX_CHOWN_RESTRICTED} | Administrative/security option (in 2.9.4) |

99 The remaining symbolic constants in 2.9.3 and 2.9.4 are useful for testing pur-
100 poses and as a guide to applications on the types of behaviors they need to be able
101 to accommodate. They do not reflect sufficient functional difference to warrant
102 requirement by Conforming POSIX.1 Applications or to distinguish between con-
103 forming implementations.

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

109 1.3.2 Application Conformance

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

113 1.3.2.1 Strictly Conforming POSIX.1 Application

114 A Strictly Conforming POSIX.1 Application is an application that requires only the
115 facilities described in this part of ISO/IEC 9945 and the applicable language stan-
116 dards. Such an application shall accept any behavior described in this part of
117 ISO/IEC 9945 as *unspecified* or *implementation-defined*, and for symbolic con-
118 stants, shall accept any value in the range permitted by this part of ISO/IEC 9945.
119 Such applications are permitted to adapt to the availability of facilities whose
120 availability is indicated by the constants in 2.8 and 2.9.

121 1.3.2.2 Conforming POSIX.1 Application

122 1.3.2.2.1 ISO/IEC Conforming POSIX.1 Application

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

1.3.2.2.2 <National Body> Conforming POSIX.1 Application

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

1.3.2.3 Conforming POSIX.1 Application Using Extensions

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

1.3.3 Language-Dependent Services for the C Programming Language

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

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

1.3.3.1 Types of Conformance

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

1.3.3.2 C Standard Language-Dependent System Support

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

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

171 1.3.3.3 Common-Usage C Language-Dependent System Support

172 Implementors, instead of referencing the C Standard (2), shall provide the rou-
173 tines and support required in Section 8 using common usage as guidance. Imple-
174 mentors shall meet all the requirements of Section 8 except where references are
175 made to the C Standard (2). In places where the C Standard (2) is referenced,
176 implementors shall provide equivalent support in a manner consistent with com-
177 mon usage of the C programming language. Implementors shall document, in
178 Section 8 of the conformance document, all differences between the interface pro-
179 vided and the interface that would have been provided had the C Standard (2)
180 been implemented instead of common usage. Implementors shall clearly docu-
181 ment the version of the C Standard (2) referenced in documenting interface differ-
182 ences and should issue updates on differences for all new versions of the
183 C Standard (2).

184

185 Where a function has been introduced by the C Standard (2), and thus there is no
186 common-usage referent for it, if the function is implemented, it shall be imple-
187 mented as described in the C Standard (2). If the function is not implemented, it
188 shall be documented as a difference from the C Standard (2) as required above.

189 1.3.4 Other C Language-Related Specifications

190 The following rules apply to the usage of C language library functions; each of the
191 statements in this subclause applies to the detailed function descriptions in Sec-
192 tions 3 through 9, unless explicitly stated otherwise:

- 193 (1) If an argument to a function has an invalid value (such as a value outside
194 the domain of the function, or a pointer outside the address space of the
195 program, or a NULL pointer when that is not explicitly permitted), the
196 behavior is undefined.
- 197 (2) Any function may also be implemented as a macro in a header. Applica-
198 tions should use #undef to remove any macro definition and ensure that
199 an actual function is referenced. Applications should also use #undef
200 prior to declaring any function in this part of ISO/IEC 9945.
- 201 (3) Any invocation of a library function that is implemented as a macro shall
202 expand to code that evaluates each of its arguments only once, fully pro-
203 tected by parentheses where necessary, so it is generally safe to use arbi-
204 trary expressions as arguments.
- 205 (4) Provided that a library function can be declared without reference to any
206 type defined in a header, it is also permissible to declare the function,
207 either explicitly or implicitly, and use it without including its associated
208 header.

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

212 **1.3.5 Other Language-Related Specifications**

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

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

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

THIS PAGE WAS
BLANK IN THE ORIGINAL

Section 2: Terminology and General Requirements

2.1 Conventions

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

(1) The *italic* font is used for:

- Cross references to defined terms within 1.3, 2.2.1, and 2.2.2; symbolic parameters that are generally substituted with real values by the application
- C language data types and function names (except in function Synopsis subclauses)
- Global external variable names

(2) The **bold** font is used with a word in all capital letters, such as

PATH

to represent an environment variable, as described in 2.6. It is also used for the term “**NULL** pointer.”

(3) The constant-width (Courier) font is used:

- For C language data types and function names within function Synopsis subclauses
- To illustrate examples of system input or output where exact usage is depicted
- For references to utility names and C language headers

(4) Symbolic constants returned by many functions as error numbers are represented as:

[ERRNO]

See 2.4.

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

{LIMIT}

See 2.8 and 2.9.

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

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

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

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

2.2 Definitions

2.2.1 Terminology

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

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

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

2.2.1.3 may: An indication of an optional feature.

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

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

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

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

2.2.1.6 should:

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

64 **2.2.1.7 supported:** A condition regarding optional functionality. |

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

71 **2.2.1.8 system documentation:** All documentation provided with an imple- |
72 mentation, except the conformance document. |

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

75 **2.2.1.9 undefined:** An indication that this part of ISO/IEC 9945 imposes no por- |
76 tability requirements on an application's use of an indeterminate value or its |
77 behavior with erroneous program constructs or erroneous data. |

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

81 **2.2.1.10 unspecified:** An indication that this part of ISO/IEC 9945 imposes no |
82 portability requirements on applications for correct program constructs or correct |
83 data regarding a value or behavior. |

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

88 **2.2.2 General Terms**

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

90 **2.2.2.1 absolute pathname:** See *pathname resolution* in 2.3.6.

91 **2.2.2.2 access mode:** A form of access permitted to a file.

92 **2.2.2.3 address space:** The memory locations that can be referenced by a
93 process.

94 **2.2.2.4 appropriate privileges:** An implementation-defined means of associat-
95 ing privileges with a process with regard to the function calls and function call
96 options defined in this part of ISO/IEC 9945 that need special privileges.

97 There may be zero or more such means.

98 **2.2.2.5 background process:** A process that is a member of a background pro-
99 cess group.

100 **2.2.2.6 background process group:** Any process group, other than a fore-
101 ground process group, that is a member of a session that has established a con-
102 nection with a controlling terminal.

103 **2.2.2.7 block special file:** A file that refers to a device.

104 A block special file is normally distinguished from a character special file by pro-
105 viding access to the device in a manner such that the hardware characteristics of
106 the device are not visible.

107 **2.2.2.8 character:** A sequence of one or more bytes representing a single
108 graphic symbol.

109 NOTE: This term corresponds in the C Standard [2] to the term *multibyte character*, noting that a
110 single-byte character is a special case of multibyte character. Unlike the usage in the C Standard
111 [2], *character* here has no necessary relationship with storage space, and *byte* is used when storage
112 space is discussed.

113 **2.2.2.9 character special file:** A file that refers to a device.

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

117 **2.2.2.10 child process:** See *process* in 2.2.2.62.

118 **2.2.2.11 clock tick:** An interval of time.

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

121 **2.2.2.12 controlling process:** The session leader that established the connec-
122 tion to the controlling terminal.

123 Should the terminal subsequently cease to be a controlling terminal for this ses-
124 sion, the session leader shall cease to be the controlling process.

125 **2.2.2.13 controlling terminal:** A terminal that is associated with a session.

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

130 **2.2.2.14 current working directory:** See *working directory* in 2.2.2.89.

- 131 **2.2.2.15 device:** A computer peripheral or an object that appears to the applica-
132 tion as such.
- 133 **2.2.2.16 directory:** A file that contains directory entries.
134 No two directory entries in the same directory shall have the same name.
- 135 **2.2.2.17 directory entry [link]:** An object that associates a filename with a file.
136 Several directory entries can associate names with the same file.
- 137 **2.2.2.18 dot:** The filename consisting of a single dot character (.).
138 See *pathname resolution* in 2.3.6.
- 139 **2.2.2.19 dot-dot:** The filename consisting solely of two dot characters (. .).
140 See *pathname resolution* in 2.3.6.
- 141 **2.2.2.20 effective group ID:** An attribute of a process that is used in determin-
142 ing various permissions, including file access permissions, described in 2.3.2.
143 See *group ID*. This value is subject to change during the process lifetime, as
144 described in 3.1.2 and 4.2.2.
- 145 **2.2.2.21 effective user ID:** An attribute of a process that is used in determining
146 various permissions, including file access permissions.
147 See *user ID*. This value is subject to change during the process lifetime, as
148 described in 3.1.2 and 4.2.2.
- 149 **2.2.2.22 empty directory:** A directory that contains, at most, directory entries
150 for dot and dot-dot.
- 151 **2.2.2.23 empty string [null string]:** A character array whose first element is a
152 null character.
- 153 **2.2.2.24 Epoch:** The time 0 hours, 0 minutes, 0 seconds, January 1, 1970, Coor-
154 dinated Universal Time.
155 See *seconds since the Epoch*.
- 156 **2.2.2.25 feature test macro:** A #defined symbol used to determine whether a
157 particular set of features will be included from a header.
158 See 2.7.1.
- 159 **2.2.2.26 FIFO special file [FIFO]:** A type of file with the property that data |
160 written to such a file is read on a first-in-first-out basis. |

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

162 **2.2.2.27 file:** An object that can be written to, or read from, or both.

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

166 **2.2.2.28 file description:** See *open file description* in 2.2.2.51.

167 **2.2.2.29 file descriptor:** A per-process unique, nonnegative integer used to
168 identify an open file for the purpose of file access.

169 **2.2.2.30 file group class:** The property of a file indicating access permissions
170 for a process related to the process's group identification.

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

175 **2.2.2.31 file mode:** An object containing the file permission bits and other
176 characteristics of a file, as described in 5.6.1.

177 **2.2.2.32 filename:** A name consisting of 1 to {NAME_MAX} bytes used to name a
178 file.

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

183 **2.2.2.33 file offset:** The byte position in the file where the next I/O operation
184 begins.

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

188 **2.2.2.34 file other class:** The property of a file indicating access permissions for
189 a process related to the process's user and group identification.

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

192 **2.2.2.35 file owner class:** The property of a file indicating access permissions
193 for a process related to the process's user identification.

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

196 **2.2.2.36 file permission bits:** Information about a file that is used, along with
197 other information, to determine if a process has read, write, or execute/search per-
198 mission to a file.

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

203 **2.2.2.37 file serial number:** A per-file system unique identifier for a file.

204 File serial numbers are unique throughout a file system.

205 **2.2.2.38 file system:** A collection of files and certain of their attributes.

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

207 **2.2.2.39 foreground process:** A process that is a member of a foreground pro-
208 cess group. |

209 **2.2.2.40 foreground process group:** A process group whose member processes |
210 have certain privileges, denied to processes in background process groups, when |
211 accessing their controlling terminal. |

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

215 **2.2.2.41 foreground process group ID:** The process group ID of the foreground
216 process group.

217 **2.2.2.42 group ID:** A nonnegative integer, which can be contained in an object of |
218 type *gid_t*, that is used to identify a group of system users. |

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

223 **2.2.2.43 job control:** A facility that allows users to selectively stop (suspend)
224 the execution of processes and continue (resume) their execution at a later point.

225 The user typically employs this facility via the interactive interface jointly sup-
226 plied by the terminal I/O driver and a command interpreter. Conforming imple-
227 mentations may optionally support job control facilities; the presence of this
228 option is indicated to the application at compile time or run time by the definition
229 of the `{_POSIX_JOB_CONTROL}` symbol; see 2.9.

230 **2.2.2.44 link:** See *directory entry* in 2.2.2.17.

- 231 **2.2.2.45 link count:** The number of directory entries that refer to a particular
232 file.
- 233 **2.2.2.46 login:** The unspecified activity by which a user gains access to the
234 system.
235 Each login shall be associated with exactly one login name.
- 236 **2.2.2.47 login name:** A user name that is associated with a login.
- 237 **2.2.2.48 mode:** A collection of attributes that specifies a file's type and its access
238 permissions.
239 See *file access permissions* in 2.3.2.
- 240 **2.2.2.49 null string:** See *empty string* in 2.2.2.23.
- 241 **2.2.2.50 open file:** A file that is currently associated with a file descriptor.
- 242 **2.2.2.51 open file description:** A record of how a process or group of processes
243 are accessing a file.
244 Each file descriptor shall refer to exactly one open file description, but an open file
245 description may be referred to by more than one file descriptor. A file offset, file
246 status (see Table 6-5), and file access modes (see Table 6-6) are attributes of an
247 open file description.
- 248 **2.2.2.52 orphaned process group:** A process group in which the parent of
249 every member is either itself a member of the group or is not a member of the
250 group's session.
- 251 **2.2.2.53 parent directory:**
252 (1) When discussing a given directory, the directory that both contains a
253 directory entry for the given directory and is represented by the path-
254 name dot-dot in the given directory.
255 (2) When discussing other types of files, a directory containing a directory
256 entry for the file under discussion.
257 This concept does not apply to dot and dot-dot.
- 258 **2.2.2.54 parent process:** See *process* in 2.2.2.62.
- 259 **2.2.2.55 parent process ID:** An attribute of a new process after it is created by
260 a currently active process.
261 The parent process ID of a process is the process ID of its creator, for the lifetime
262 of the creator. After the creator's lifetime has ended, the parent process ID is the
263 process ID of an implementation-defined system process.

264 **2.2.2.56 path prefix:** A pathname, with an optional ending slash, that refers to
265 a directory.

266 **2.2.2.57 pathname:** A string that is used to identify a file.

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

275 **2.2.2.58 pathname component:** See *filename* in 2.2.2.32.

276 **2.2.2.59 pipe:** An object accessed by one of the pair of file descriptors created by
277 the *pipe()* function.

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

281 **2.2.2.60 portable filename character set:** The set of characters from which
282 portable filenames are constructed.

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

```
285     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
286     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
287     0 1 2 3 4 5 6 7 8 9 . _ -
```

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

293 **2.2.2.61 privilege:** See *appropriate privileges* in 2.2.2.4.

294 **2.2.2.62 process:** An address space and single thread of control that executes
295 within that address space, and its required system resources.

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

299 **2.2.2.63 process group:** A collection of processes that permits the signaling of
300 related processes.

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

303 **2.2.2.64 process group ID:** The unique identifier representing a process group |
304 during its lifetime. |

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

307 **2.2.2.65 process group leader:** A process whose process ID is the same as its
308 process group ID.

309 **2.2.2.66 process group lifetime:** A period of time that begins when a process
310 group is created and ends when the last remaining process in the group leaves the
311 group, due either to the end of the last process's process lifetime or to the last
312 remaining process calling the *setsid()* or *setpgid()* functions. |

313 **2.2.2.67 process ID:** The unique identifier representing a process. |

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

319 **2.2.2.68 process lifetime:** The period of time that begins when a process is |
320 created and ends when its process ID is returned to the system. |

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

328 **2.2.2.69 read-only file system:** A file system that has implementation-defined
329 characteristics restricting modifications.

330 **2.2.2.70 real group ID:** The attribute of a process that, at the time of process
331 creation, identifies the group of the user who created the process.

332 See *group ID* in 2.2.2.42. This value is subject to change during the process life-
333 time, as described in 4.2.2.

334 **2.2.2.71 real user ID:** The attribute of a process that, at the time of process
335 creation, identifies the user who created the process.

336 See *user ID* in 2.2.2.87. This value is subject to change during the process life-
337 time, as described in 4.2.2.

338 **2.2.2.72 regular file:** A file that is a randomly accessible sequence of bytes, with
339 no further structure imposed by the system.

340 **2.2.2.73 relative pathname:** See *pathname resolution* in 2.3.6.

341 **2.2.2.74 root directory:** A directory, associated with a process, that is used in
342 pathname resolution for pathnames that begin with a slash.

343 **2.2.2.75 saved set-group-ID:** An attribute of a process that allows some flexibil-
344 ity in the assignment of the effective group ID attribute, when the saved set-user-
345 ID option is implemented, as described in 3.1.2 and 4.2.2.

346 **2.2.2.76 saved set-user-ID:** An attribute of a process that allows some flexibil-
347 ity in the assignment of the effective user ID attribute, when the saved set-user-ID
348 option is implemented, as described in 3.1.2 and 4.2.2.

349 **2.2.2.77 seconds since the Epoch:** A value to be interpreted as the number of
350 seconds between a specified time and the Epoch.

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

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

358
$$tm_sec + tm_min*60 + tm_hour*3\,600 + tm_yday*86\,400 +$$

359
$$(tm_year-70)*31\,536\,000 + ((tm_year-69)/4)*86\,400$$

360 **2.2.2.78 session:** A collection of process groups established for job control
361 purposes.

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

367 **2.2.2.79 session leader:** A process that has created a session (see 4.3.2).

368 **2.2.2.80 session lifetime:** The period between when a session is created and the
369 end of the lifetime of all the process groups that remain as members of the
370 session.

371 **2.2.2.81 signal:** A mechanism by which a process may be notified of, or affected
372 by, an event occurring in the system.

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

375 **2.2.2.82 slash:** The literal character “/”.

376 This character is also known as *solidus* in ISO 8859-1 (B34).

377 **2.2.2.83 supplementary group ID:** An attribute of a process used in determin- |
378 ing file access permissions. |

379 A process has up to (NGROUPS_MAX) supplementary group IDs in addition to the
380 effective group ID. The supplementary group IDs of a process are set to the sup-
381plementary group IDs of the parent process when the process is created. Whether
382a process’s effective group ID is included in or omitted from its list of supplemen-
383tary group IDs is unspecified.

384 **2.2.2.84 system:** An implementation of this part of ISO/IEC 9945.

385 **2.2.2.85 system process:** An object, other than a process executing an applica-
386tion, that is defined by the system and has a process ID.

387 **2.2.2.86 terminal [terminal device]:** A character special file that obeys the
388 specifications of 7.1. |

389 **2.2.2.87 user ID:** A nonnegative integer, which can be contained in an object of |
390 type *uid_t*, that is used to identify a system user. |

391 When the identity of a user is associated with a process, a user ID value is
392referred to as a real user ID, an effective user ID, or an (optional) saved
393set-user-ID.

394 **2.2.2.88 user name:** A string that is used to identify a user, as described in 9.1. |

395 **2.2.2.89 working directory [current working directory]:** A directory, asso-
396ciated with a process, that is used in pathname resolution for pathnames that do
397not begin with a slash.

398 **2.2.3 Abbreviations** |

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

400 **2.2.3.1 C Standard:** ISO/IEC 9899, *Information technology—Programming* |
401 *languages—C* (2). |

402 **2.2.3.2 IRV:** The International Reference Version coded character set described |
403 in ISO/IEC 646 (1). |

404 **2.2.3.3 POSIX.1:** This part of ISO/IEC 9945. |

405 **2.3 General Concepts**

406 **2.3.1 extended security controls:** The access control (see *file access permis-*
407 *sions*) and privilege (see *appropriate privileges* in 2.2.2.4) mechanisms have been
408 defined to allow implementation-defined extended security controls. These permit
409 an implementation to provide security mechanisms to implement different secu-
410 rity policies than described in this part of ISO/IEC 9945. These mechanisms shall
411 not alter or override the defined semantics of any of the functions in this part of
412 ISO/IEC 9945.

413 **2.3.2 file access permissions:** The standard file access control mechanism uses
414 the file permission bits, as described below. These bits are set at file creation by
415 *open()*, *creat()*, *mkdir()*, and *mkfifo()* and are changed by *chmod()*. These bits are
416 read by *stat()* or *fstat()*.

417 Implementations may provide *additional* or *alternate* file access control mechan-
418 isms, or both. An additional access control mechanism shall only further restrict
419 the access permissions defined by the file permission bits. An alternate access
420 control mechanism shall:

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

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

- 432 (1) If a process has the appropriate privilege:
 - 433 (a) If read, write, or directory search permission is requested, access is
434 granted.
 - 435 (b) If execute permission is requested, access is granted if execute per-
436 mission is granted to at least one user by the file permission bits or
437 by an alternate access control mechanism; otherwise, access is
438 denied.
- 439 (2) Otherwise:
 - 440 (a) The file permission bits of a file contain read, write, and
441 execute/search permissions for the file owner class, file group class,
442 and file other class.
 - 443 (b) Access is granted if an alternate access control mechanism is not |
444 enabled and the requested access permission bit is set for the class |

445 (file owner class, file group class, or file other class) to which the
446 process belongs, or if an alternate access control mechanism is
447 enabled and it allows the requested access; otherwise, access is
448 denied.

449 **2.3.3 file hierarchy:** Files in the system are organized in a hierarchical struc-
450 ture in which all of the nonterminal nodes are directories and all of the terminal
451 nodes are any other type of file. Because multiple directory entries may refer to
452 the same file, the hierarchy is properly described as a “directed graph.”

453 **2.3.4 filename portability:** Filenames should be constructed from the portable
454 filename character set because the use of other characters can be confusing or
455 ambiguous in certain contexts.

456 **2.3.5 file times update:** Each file has three distinct associated time values:
457 *st_atime*, *st_mtime*, and *st_ctime*. The *st_atime* field is associated with the times
458 that the file data is accessed; *st_mtime* is associated with the times that the file
459 data is modified; and *st_ctime* is associated with the times that file status is
460 changed. These values are returned in the file characteristics structure, as
461 described in 5.6.1.

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

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

476 **2.3.6 pathname resolution:** Pathname resolution is performed for a process to
477 resolve a pathname to a particular file in a file hierarchy. There may be multiple
478 pathnames that resolve to the same file.

479 Each filename in the pathname is located in the directory specified by its prede-
480 cessor (for example, in the pathname fragment “a/b”, file “b” is located in direc-
481 tory “a”). Pathname resolution fails if this cannot be accomplished. If the path-
482 name begins with a slash, the predecessor of the first filename in the pathname is
483 taken to be the root directory of the process (such pathnames are referred to as
484 absolute pathnames). If the pathname does not begin with a slash, the predeces-
485 sor of the first filename of the pathname is taken to be the current working direc-
486 tory of the process (such pathnames are referred to as “relative pathnames”).

487 The interpretation of a pathname component is dependent on the values of
488 {NAME_MAX} and {_POSIX_NO_TRUNC} associated with the path prefix of that
489 component. If any pathname component is longer than {NAME_MAX}, and

490 {`_POSIX_NO_TRUNC`} is in effect for the path prefix of that component (see 5.7.1),
 491 the implementation shall consider this an error condition. Otherwise, the imple-
 492 mentation shall use the first {`NAME_MAX`} bytes of the pathname component.

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

497 A pathname consisting of a single slash resolves to the root directory of the pro-
 498 cess. A null pathname is invalid.

499 2.4 Error Numbers

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

```
502     extern int errno;
```

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

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

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

521 Implementations may generate error numbers listed in this clause under cir-
 522 cumstances other than those described, if and only if all those error conditions can
 523 always be treated identically to the error conditions as described in this part of
 524 ISO/IEC 9945. Implementations may support additional errors not listed in this
 525 clause, but shall not generate a different error number from one required by this
 526 part of ISO/IEC 9945 for an error condition described in this part of ISO/IEC 9945.

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

533	The actual values are unspecified by this part of ISO/IEC 9945.	
534	[E2BIG]	Arg list too long
535		The sum of the number of bytes used by the new process image's
536		argument list and environment list was greater than the system-
537		imposed limit of {ARG_MAX} bytes.
538	[EACCES]	Permission denied
539		An attempt was made to access a file in a way forbidden by its file
540		access permissions.
541	[EAGAIN]	Resource temporarily unavailable
542		This is a temporary condition, and later calls to the same routine
543		may complete normally.
544	[EBADF]	Bad file descriptor
545		A file descriptor argument was out of range, referred to no open
546		file, or a read (write) request was made to a file that was only
547		open for writing (reading).
548	[EBUSY]	Resource busy
549		An attempt was made to use a system resource that was not
550		available at the time because it was being used by a process in a
551		manner that would have conflicted with the request being made
552		by this process.
553	[ECHILD]	No child processes
554		A <i>wait()</i> or <i>waitpid()</i> function was executed by a process that had
555		no existing or unwaited-for child processes.
556	[EDEADLK]	Resource deadlock avoided
557		An attempt was made to lock a system resource that would have
558		resulted in a deadlock situation.
559	[EDOM]	Domain error
560		Defined in the C Standard (2); an input argument was outside the
561		defined domain of the mathematical function.
562	[EEXIST]	File exists
563		An existing file was specified in an inappropriate context; for
564		instance, as the new link name in a <i>link()</i> function.
565	[EFAULT]	Bad address
566		The system detected an invalid address in attempting to use an
567		argument of a call. The reliable detection of this error is imple-
568		mentation defined; however, implementations that do detect this
569		condition shall use this value.
570	[EFBIG]	File too large
571		The size of a file would exceed an implementation-defined max-
572		imum file size.
573	[EINTR]	Interrupted function call
574		An asynchronous signal (such as SIGINT or SIGQUIT; see the
575		description of header <signal.h> in 3.3.1) was caught by the
576		process during the execution of an interruptible function. If the

577		signal handler performs a normal return, the interrupted function call may return this error condition.
578		
579	[EINVAL]	Invalid argument
580		Some invalid argument was supplied. [For example, specifying an undefined signal to a <i>signal()</i> or <i>kill()</i> function].
581		
582	[EIO]	Input/output error
583		Some physical input or output error occurred. This error may be reported on a subsequent operation on the same file descriptor.
584		Any other error-causing operation on the same file descriptor may cause the [EIO] error indication to be lost.
585		
586		
587	[EISDIR]	Is a directory
588		An attempt was made to open a directory with write mode specified.
589		
590	[EMFILE]	Too many open files
591		An attempt was made to open more than the maximum number of {OPEN_MAX} file descriptors allowed in this process.
592		
593	[EMLINK]	Too many links
594		An attempt was made to have the link count of a single file exceed {LINK_MAX}.
595		
596	[ENAMETOOLONG]	Filename too long
597		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.
598		
599		
600	[ENFILE]	Too many open files in system
601		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.
602		
603		
604	[ENODEV]	No such device
605		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.
606		
607		
608	[ENOENT]	No such file or directory
609		A component of a specified pathname did not exist, or the pathname was an empty string.
610		
611	[ENOEXEC]	Exec format error
612		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.
613		
614		
615	[ENOLCK]	No locks available
616		A system-imposed limit on the number of simultaneous file and record locks was reached, and no more were available at that time.
617		
618		
619	[ENOMEM]	Not enough space
620		The new process image required more memory than was allowed

621		by the hardware or by system-imposed memory management
622		constraints.
623	[ENOSPC]	No space left on device
624		During a <i>write()</i> function on a regular file, or when extending a
625		directory, there was no free space left on the device.
626	[ENOSYS]	Function not implemented
627		An attempt was made to use a function that is not available in
628		this implementation.
629	[ENOTDIR]	Not a directory
630		A component of the specified pathname existed, but it was not a
631		directory, when a directory was expected.
632	[ENOTEMPTY]	Directory not empty
633		A directory with entries other than dot and dot-dot was supplied
634		when an empty directory was expected.
635	[ENOTTY]	Inappropriate I/O control operation
636		A control function was attempted for a file or a special file for
637		which the operation was inappropriate.
638	[ENXIO]	No such device or address
639		Input or output on a special file referred to a device that did not
640		exist, or made a request beyond the limits of the device. This
641		error may also occur when, for example, a tape drive is not online
642		or a disk pack is not loaded on a drive.
643	[EPERM]	Operation not permitted
644		An attempt was made to perform an operation limited to
645		processes with appropriate privileges or to the owner of a file or
646		other resource.
647	[EPIPE]	Broken pipe
648		A write was attempted on a pipe or FIFO for which there was no
649		process to read the data.
650	[ERANGE]	Result too large
651		Defined in the C Standard [2]; the result of the function was too
652		large to fit in the available space.
653	[EROFS]	Read-only file system
654		An attempt was made to modify a file or directory on a file system
655		that was read-only at that time.
656	[ESPIPE]	Invalid seek
657		An <i>lseek()</i> function was issued on a pipe or FIFO.
658	[ESRCH]	No such process
659		No process could be found corresponding to that specified by the
660		given process ID.
661	[EXDEV]	Improper link
662		A link to a file on another file system was attempted.

2.5 Primitive System Data Types

Some data types used by the various system functions are not defined as part of this part of ISO/IEC 9945, but are defined by the implementation. These types are then defined in the header `<sys/types.h>`, which contains definitions for at least the types shown in Table 2-1.

Table 2-1 – Primitive System Data Types

Defined Type	Description
<i>dev_t</i>	Used for device numbers.
<i>gid_t</i>	Used for group IDs.
<i>ino_t</i>	Used for file serial numbers.
<i>mode_t</i>	Used for some file attributes, for example file type, file access permissions.
<i>nlink_t</i>	Used for link counts.
<i>off_t</i>	Used for file sizes.
<i>pid_t</i>	Used for process IDs and process group IDs.
<i>size_t</i>	As defined in the C Standard [2].
<i>ssize_t</i>	Used by functions that return a count of bytes (memory space) or an error indication.
<i>uid_t</i>	Used for user IDs.

All of the types listed in Table 2-1 shall be arithmetic types; *pid_t*, *ssize_t*, and *off_t* shall be signed arithmetic types. The type *ssize_t* shall be capable of storing values in the range from `-1` to `{SSIZE_MAX}`, inclusive. The types *size_t* and *ssize_t* shall also be defined in the header `<unistd.h>`.

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

2.6 Environment Description

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

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

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

HOME	The name of the user’s initial working directory from the user database (see the description of the header <code><pwd.h></code> in 9.2.2).
-------------	--

703	LANG	The name of the locale to use for locale categories when both LC_ALL and the corresponding environment variable (beginning with “ LC_ ”) do not specify a locale.
704		
705		
706	LC_ALL	The name of the locale to be used to override any values for locale categories specified by the setting of LANG or any environment variables beginning with “ LC_ ”.
707		
708		
709	LC_COLLATE	The name of the locale for collation information.
710	LC_CTYPE	The name of the locale for character classification.
711	LC_MONETARY	The name of the locale containing monetary-related numeric editing information.
712		
713	LC_NUMERIC	The name of the locale containing numeric editing (i.e., radix character) information.
714		
715	LC_TIME	The name of the locale for date/time formatting information.
716		
717	LOGNAME	The login name associated with the current process. The value shall be composed of characters from the portable filename character set.
718		
719		
720		NOTE: An application that requires, or an installation that actually uses, characters outside the portable filename character set would not strictly conform to this part of ISO/IEC 9945. However, it is reasonable to expect that such characters would be used in many countries (recognizing the reduced level of interchange implied by this), and applications or installations should permit such usage where possible. No error is defined by this part of ISO/IEC 9945 for violation of this condition.
721		
722		
723		
724		
725		
726		
727		
728	PATH	The sequence of path prefixes that certain functions apply in searching for an executable file known only by a filename (a pathname that does not contain a slash). The prefixes are separated by a colon (:). When a nonzero-length prefix is applied to this filename, a slash is inserted between the prefix and the filename. A zero-length prefix is a special prefix that indicates the current working directory. It appears as two adjacent colons (::), as an initial colon preceding the rest of the list, or as a trailing colon following the rest of the list. The list is searched from beginning to end until an executable program by the specified name is found. If the pathname being sought contains a slash, the search through the path prefixes is not performed.
729		
730		
731		
732		
733		
734		
735		
736		
737		
738		
739		
740		
741		
742	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.
743		
744		
745		

746 **TZ** Time zone information. The format of this string is
747 defined in 8.1.1.

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

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

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

761 **2.7 C Language Definitions**

762 **2.7.1 Symbols From the C Standard**

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

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

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

- 772 (1) All external identifiers that begin with an underscore are reserved.
- 773 (2) All other identifiers that begin with an underscore and either an upper-
774 case letter or another underscore are reserved.
- 775 (3) If the program defines an external identifier with the same name as a
776 reserved external identifier, even in a semantically equivalent form, the
777 behavior is undefined.

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

783 2.7.2 POSIX.1 Symbols

784 Certain symbols in this part of ISO/IEC 9945 are defined in headers. Some of
785 those headers could also define other symbols than those defined by this part of
786 ISO/IEC 9945, potentially conflicting with symbols used by the application. Also,
787 this part of ISO/IEC 9945 defines symbols that are not permitted by other stan-
788 dards to appear in those headers without some control on the visibility of those
789 symbols.

790 Symbols called *feature test macros* are used to control the visibility of symbols
791 that might be included in a header. Implementations, future versions of this part
792 of ISO/IEC 9945, and other standards may define additional feature test macros.
793 Feature test macros shall be defined in the compilation of an application before an
794 `#include` of any header where a symbol should be visible to some, but not all,
795 applications. If the definition of the macro does not precede the `#include`, the
796 result is undefined.

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

798 Implementations may add symbols to the headers shown in Table 2-2, provided
799 the identifiers for those symbols begin with the corresponding reserved prefixes in
800 Table 2-2. Similarly, implementations may add symbols to the headers in
801 Table 2-2 that end in the string indicated as a reserved suffix as long as the
802 reserved suffix is in that part of the name considered significant by the implemen-
803 tation. This shall be in addition to any reservations made in the C Standard [2].

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

807 After the last inclusion of a given header, an application may use any of the sym-
808 bol classes reserved in Table 2-2 for its own purposes, as long as the requirements
809 in the note to Table 2-2 are satisfied, noting that the symbol declared in the
810 header may become inaccessible.

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

813 In addition, implementations may add members to a structure or union without
814 controlling the visibility of those members with a feature test macro, as long as a
815 user-defined macro with the same name cannot interfere with the correct
816 interpretation of the program.

817 The header `<fcntl.h>` may contain the following symbols in addition to those
818 specifically required elsewhere in POSIX.1:

819	SEEK_CUR	S_IRUSR	S_ISCHR	S_ISREG	S_IWUSR
820	SEEK_END	S_IRWXG	S_ISDIR	S_ISUID	S_IXGRP
821	SEEK_SET	S_IRWXO	S_ISFIFO	S_IWGRP	S_IXOTH
822	S_IRGRP	S_IRWXU	S_ISGID	S_IWOTH	S_IXUSR
823	S_IROTH	S_ISBLK			

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

Table 2-2 – Reserved Header Symbols

Header	Key	Reserved Prefix	Reserved Suffix
<dirent.h>	1	d_	
<fcntl.h>	1	l_	
	2	F_	
	2	O_	
	2	S_	
<grp.h>	1	gr_	
<limits.h>	1		_MAX
<locale.h>	2	LC_[A-Z]	
<pwd.h>	1	pw_	
<signal.h>	1	sa_	
	2	SIG_	
	2	SA_	
<sys/stat.h>	1	st_	
	2	S_	
<sys/times.h>	1	tms_	
<termios.h>	1	c_	
	2	V_	
	2	I_	
	2	O_	
	2	TC_	
	2	B[0-9]	
any POSIX.1 header included	1		_t

NOTE: The notation “[0-9]” indicates any digit and “[A-Z]” any uppercase character in the portable filename character set. The Key values are:

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

The following feature test macro is defined:

Name	Description
_POSIX_SOURCE	When an application includes a header described by POSIX.1, and when this feature test macro is defined according to the preceding rules: <ol style="list-style-type: none"> (1) All symbols required by POSIX.1 to appear when the header is included shall be made visible. (2) Symbols that are explicitly permitted, but not required, by POSIX.1 to appear in that header (including those in reserved namespaces) may be made visible. (3) Additional symbols not required or explicitly permitted by POSIX.1 to be in that header shall not be made visible.

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

875 **2.7.2.1 C Standard Language-Dependent Support**

876 If there are no feature test macros present in a program, the implementation
877 shall make visible only those identifiers specified as reserved identifiers in the
878 C Standard {2}, permitting the reservation of symbols and namespace defined in
879 2.7.1. For each feature test macro present, only the symbols specified by that
880 feature test macro plus those of the C Standard {2} shall be defined when a header
881 is included.

882 **2.7.2.2 Common-Usage-Dependent Support**

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

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

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

894 **2.7.3 Headers and Function Prototypes**

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

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

899 For functions described in the C Standard {2} and included by reference in Section
900 8 (whether or not they are further described in this part of ISO/IEC 9945), these
901 prototypes or declarations (if required) shall appear in the headers defined for
902 them in the C Standard {2}. For other functions in this part of ISO/IEC 9945, the
903 prototypes or declarations shall appear in the headers listed below. If a function
904 is defined by this part of ISO/IEC 9945, is not described in the C Standard {2}, and
905 is not listed below, it shall have its prototype or declaration (if required) appear in
906 `<unistd.h>`, which shall be `#include`-ed by the application before using any
907 function declared in it, whether or not it is mentioned in the Synopsis subclause
908 for that function. The requirements about the visibility of symbols in 2.7.2 shall
909 be honored.

910	<dirent.h>	<i>opendir()</i> , <i>readdir()</i> , <i>rewinddir()</i> , <i>closedir()</i>	
911	<fcntl.h>	<i>open()</i> , <i>creat()</i> , <i>fcntl()</i>	
912	<grp.h>	<i>getgrgid()</i> , <i>getgrnam()</i>	
913	<pwd.h>	<i>getpwuid()</i> , <i>getpwnam()</i>	
914	<setjmp.h>	<i>sigsetjmp()</i> , <i>siglongjmp()</i>	
915	<signal.h>	<i>kill()</i> , <i>sigemptyset()</i> , <i>sigfillset()</i> , <i>sigaddset()</i> , <i>sigdelset()</i> , 916 <i>sigismember()</i> , <i>sigaction()</i> , <i>sigprocmask()</i> , <i>sigpending()</i> , 917 <i>sigsuspend()</i>	
918	<stdio.h>	<i>ctermid()</i> , <i>fileno()</i> , <i>fdopen()</i>	
919	<sys/stat.h>	<i>umask()</i> , <i>mkdir()</i> , <i>mkfifo()</i> , <i>stat()</i> , <i>fstat()</i> , <i>chmod()</i>	
920	<sys/times.h>	<i>times()</i>	
921	<sys/utsname.h>	<i>uname()</i>	
922	<sys/wait.h>	<i>wait()</i> , <i>waitpid()</i>	
923	<termios.h>	<i>cfgetospeed()</i> , <i>cfsetospeed()</i> , <i>cfgetispeed()</i> , <i>cfsetispeed()</i> , 924 <i>tcgetattr()</i> , <i>tcsetattr()</i> , <i>tcsendbreak()</i> , <i>tcdrain()</i> , 925 <i>tcflush()</i> , <i>tcflow()</i>	
926	<time.h>	<i>time()</i> , <i>tzset()</i>	
927	<utime.h>	<i>utime()</i>	

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

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

940 Neither the names of the formal parameters nor their types, as they appear in an
941 implementation, are specified by this part of ISO/IEC 9945. The names are used
942 within this part of ISO/IEC 9945 as a notational mechanism. However, any
943 declaration provided by an implementation shall accept all actual parameter
944 types that a declaration lexically identical to one in this part of ISO/IEC 9945 shall
945 accept, including the effects of both type conversion and checking for the number
946 of arguments implied by the presence of a filled-out prototype. The
947 implementation's declaration shall not cause a syntax error if an application pro-
948 vides a prototype lexically identical to one in this part of ISO/IEC 9945. It is not a
949 requirement that nonconforming parameters to functions that may be used by an
950 application be diagnosed by an implementation, except as specifically required by
951 this part of ISO/IEC 9945 or the C Standard [2], as applicable. Where the

952 C Standard {2} has a more restrictive requirement for a function defined by that
953 standard, that requirement shall be honored, and this exception does not apply.

954 2.8 Numerical Limits

955 The following subclauses list magnitude limitations imposed by a specific imple-
956 mentation. The braces notation, {LIMIT}, is used in this part of ISO/IEC 9945 to
957 indicate these values, but the braces are not part of the name.

958 2.8.1 C Language Limits

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

964 2.8.2 Minimum Values

965 The symbols in Table 2-3 shall be defined in `<limits.h>` with the values shown.
966 These are symbolic names for the most restrictive value for certain features on a
967 system conforming to this part of ISO/IEC 9945. Related symbols are defined else-
968 where in this part of ISO/IEC 9945, which reflect the actual implementation and
969 which need not be as restrictive. A conforming implementation shall provide
970 values at least this large. A portable application shall not require a larger value
971 for correct operation.

972 2.8.3 Run-Time Increaseable Values

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

974 A Strictly Conforming POSIX.1 Application shall assume that the value supplied
975 by `<limits.h>` in a specific implementation is the minimum value that pertains
976 whenever the Strictly Conforming POSIX.1 Application is run under that imple-
977 mentation.³⁾ A specific instance of a specific implementation may increase the
978 value relative to that supplied by `<limits.h>` for that implementation. The
979 actual value supported by a specific instance shall be provided by the `sysconf()`
980 function.

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

Table 2-3 – Minimum Values

Name	Description	Value
{_POSIX_ARG_MAX}	The length of the arguments for one of the <i>exec</i> functions, in bytes, including environment data.	4096
{_POSIX_CHILD_MAX}	The number of simultaneous processes per real user ID.	6
{_POSIX_LINK_MAX}	The value of a file's link count.	8
{_POSIX_MAX_CANON}	The number of bytes in a terminal canonical input queue.	255
{_POSIX_MAX_INPUT}	The number of bytes for which space will be available in a terminal input queue.	255
{_POSIX_NAME_MAX}	The number of bytes in a filename.	14
{_POSIX_NGROUPS_MAX}	The number of simultaneous supplementary group IDs per process.	0
{_POSIX_OPEN_MAX}	The number of files that one process can have open at one time.	16
{_POSIX_PATH_MAX}	The number of bytes in a pathname.	255
{_POSIX_PIPE_BUF}	The number of bytes that can be written atomically when writing to a pipe.	512
{_POSIX_SSIZE_MAX}	The value that can be stored in an object of type <i>ssize_t</i> .	32 767
{_POSIX_STREAM_MAX}	The number of streams that one process can have open at one time.	8
{_POSIX_TZNAME_MAX}	The maximum number of bytes supported for the name of a time zone (not of the TZ variable).	3

Table 2-4 – Run-Time Increasable Values

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

2.8.4 Run-Time Invariant Values (Possibly Indeterminate)

A definition of one of the values in Table 2-5 shall be omitted from the `<limits.h>` on specific implementations where the corresponding value is equal to or greater than the stated minimum, but is indeterminate.

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

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

Name	Description	Minimum Value
{ARG_MAX}	Maximum length of arguments for the <i>exec</i> functions, in bytes, including environment data.	{_POSIX_ARG_MAX}
{CHILD_MAX}	Maximum number of simultaneous processes per real user ID.	{_POSIX_CHILD_MAX}
{OPEN_MAX}	Maximum number of files that one process can have open at any given time.	{_POSIX_OPEN_MAX}
{STREAM_MAX}	The number of streams that one process can have open at one time. If defined, it shall have the same value as {FOPEN_MAX} from the C Standard (2).	{_POSIX_STREAM_MAX}
{TZNAME_MAX}	The maximum number of bytes supported for the name of a time zone (not of the <i>TZ</i> variable).	{_POSIX_TZNAME_MAX}

2.8.5 Pathname Variable Values

The values in Table 2-6 may be constants within an implementation or may vary from one pathname to another.

Table 2-6 – Pathname Variable Values

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

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

A definition of one of the values from Table 2-6 shall be omitted from `<limits.h>` on specific implementations where the corresponding value is equal to or greater than the stated minimum, but where the value can vary depending

1068 on the file to which it is applied. The actual value supported for a specific path-
1069 name shall be provided by the *pathconf()* function.

1070 2.8.6 Invariant Values

1071 The value in Table 2-7 shall not vary in a given implementation. The value in
1072 that table shall appear in `<limits.h>`.

1073 **Table 2-7 – Invariant Value**

1074	Name	Description	Value
1075	{SSIZE_MAX}	The maximum value that can be stored in an 1076 object of type <i>ssize_t</i> . 1077	{_POSIX_SSIZE_MAX}
1078			

1079 2.9 Symbolic Constants

1080 A conforming implementation shall have the header `<unistd.h>`. This header
1081 defines the symbolic constants and structures referenced elsewhere in this part of
1082 ISO/IEC 9945. The constants defined by this header are shown in the following
1083 subclauses. The actual values of the constants are implementation defined.

1084 2.9.1 Symbolic Constants for the *access()* Function

1085 The constants used by the *access()* function are shown in Table 2-8. The con-
1086 stants *F_OK*, *R_OK*, *W_OK*, and *X_OK*, and the expressions

1087 *R_OK* | *W_OK*

1088 (where the | represents the bitwise inclusive OR operator),

1089 *R_OK* | *X_OK*

1090 and

1091 *R_OK* | *W_OK* | *X_OK*

1092 shall all have distinct values.

1093 2.9.2 Symbolic Constant for the *lseek()* Function

1094 The constants used by the *lseek()* function are shown in Table 2-9.

Table 2-8 – Symbolic Constants for the *access()* Function

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

Table 2-9 – Symbolic Constants for the *lseek()* Function

Constant	Description
SEEK_SET	Set file offset to <i>offset</i> .
SEEK_CUR	Set file offset to current plus <i>offset</i> .
SEEK_END	Set file offset to EOF plus <i>offset</i> .

2.9.3 Compile-Time Symbolic Constants for Portability Specifications

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

Table 2-10 – Compile-Time Symbolic Constants

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

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

2.9.4 Execution-Time Symbolic Constants for Portability Specifications

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

Table 2-11 – Execution-Time Symbolic Constants

	Name	Description
1136	<code>[_POSIX_CHOWN_RESTRICTED]</code>	The use of the <i>chown()</i> 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.
1137		
1138	<code>[_POSIX_NO_TRUNC]</code>	Pathname components longer than <code>(NAME_MAX)</code> generate an error.
1139		
1140	<code>[_POSIX_VDISABLE]</code>	Terminal special characters defined in 7.1.1.9 can be disabled using this character value, if it is defined. See <i>tcgetattr()</i> and <i>tcsetattr()</i> .
1141		
1142		
1143		
1144		
1145		

1146 If any of the constants in Table 2-11 are not defined in the header `<unistd.h>`,
 1147 the value varies depending on the file to which it is applied. See 5.7.1.

1148 If any of the constants in Table 2-11 are defined to have value `-1` in the header
 1149 `<unistd.h>`, the implementation shall not provide the option on any file; if any
 1150 are defined to have a value other than `-1` in the header `<unistd.h>`, the imple-
 1151 mentation shall provide the option on all applicable files.

1152 All of the constants in Table 2-11, whether defined in `<unistd.h>` or not, may be
 1153 queried with respect to a specific file using the *pathconf()* or *fpathconf()* functions.

THIS PAGE WAS
BLANK IN THE ORIGINAL

Section 3: Process Primitives

1 The functions described in this section perform the most primitive operating sys- |
2 tem services dealing with processes, interprocess signals, and timers. All attri-
3 butes of a process that are specified in this part of ISO/IEC 9945 shall remain
4 unchanged by a process primitive unless the description of that process primitive
5 states explicitly that the attribute is changed.

6 3.1 Process Creation and Execution

7 3.1.1 Process Creation

8 Function: *fork()*

9 3.1.1.1 Synopsis

10 #include <sys/types.h>
11 pid_t fork(void); |

12 3.1.1.2 Description

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

- 15 (1) The child process has a unique process ID. The child process ID also does
16 not match any active process group ID.
- 17 (2) The child process has a different parent process ID (which is the process
18 ID of the parent process).
- 19 (3) The child process has its own copy of the parent's file descriptors. Each
20 of the child's file descriptors refers to the same open file description with
21 the corresponding file descriptor of the parent.
- 22 (4) The child process has its own copy of the parent's open directory streams
23 (see 5.1.2). Each open directory stream in the child process may share
24 directory stream positioning with the corresponding directory stream of
25 the parent.
- 26 (5) The child process's values of *tms_utime*, *tms_stime*, *tms_cutime*, and
27 *tms_cstime* are set to zero (see 4.5.2).

- 28 (6) File locks previously set by the parent are not inherited by the child.
29 (See 6.5.2.)
- 30 (7) Pending alarms are cleared for the child process. (See 3.4.1.)
- 31 (8) The set of signals pending for the child process is initialized to the empty
32 set. (See 3.3.1.)

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

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

39 3.1.1.3 Returns

40 Upon successful completion, *fork()* shall return a value of zero to the child process
41 and shall return the process ID of the child process to the parent process. Both
42 processes shall continue to execute from the *fork()* function. Otherwise, a value of
43 -1 shall be returned to the parent process, no child process shall be created, and
44 *errno* shall be set to indicate the error.

45 3.1.1.4 Errors

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

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

51 For each of the following conditions, if the condition is detected, the *fork()* func-
52 tion shall return -1 and set *errno* to the corresponding value:

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

55 3.1.1.5 Cross-References

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

57 3.1.2 Execute a File

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

59 3.1.2.1 Synopsis

60 int execl(const char *path, const char *arg, ...);
61 int execv(const char *path, char *const argv[]);

```

62  int execl(const char *path, const char *arg, ...);
63  int execve(const char *path, char *const argv[], char *const envp[]);
64  int execlp(const char *file, const char *arg, ...);
65  int execvp(const char *file, char *const argv[]);

```

66 3.1.2.2 Description

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

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

```

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

```

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

```

76     extern char **environ;

```

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

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

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

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

90
 91 The argument *argv* is an array of character pointers to null-terminated strings.
 92 The last member of this array shall be a NULL pointer. These strings constitute
 93 the argument list available to the new process image. The value in *argv*[0] should
 94 point to a filename that is associated with the process being started by one of the
 95 *exec* functions.

96 The const char **arg* and subsequent ellipses in the *execl()*, *execlp()*, and *exe-*
 97 *cle()* functions can be thought of as *arg0*, *arg1*, ..., *argn*. Together they describe
 98 a list of one or more pointers to null-terminated character strings that represent
 99 the argument list available to the new program. The first argument should point
 100 to a filename that is associated with the process being started by one of the *exec*
 101 functions, and the last argument shall be a NULL pointer. For the *execle()* func-
 102 tion, the environment is provided by following the NULL pointer that shall ter-
 103 minate the list of arguments in the parameter list to *execle()* with an additional

104 parameter, as if it were declared as

105 `char *const envp[]`

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

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

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

117 File descriptors open in the calling process image remain open in the new process
118 image, except for those whose close-on-exec flag FD_CLOEXEC is set (see 6.5.2 and
119 6.5.1). For those file descriptors that remain open, all attributes of the open file
120 description, including file locks (see 6.5.2), remain unchanged by this function
121 call.

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

124 Signals set to the default action (SIG_DFL) in the calling process image shall be
125 set to the default action in the new process image. Signals set to be ignored
126 (SIG_IGN) by the calling process image shall be set to be ignored by the new pro-
127 cess image. Signals set to be caught by the calling process image shall be set to
128 the default action in the new process image (see 3.3.1).

129 If the set-user-ID mode bit of the new process image file is set (see 5.6.4), the effec-
130 tive user ID of the new process image is set to the owner ID of the new process
131 image file. Similarly, if the set-group-ID mode bit of the new process image file is
132 set, the effective group ID of the new process image is set to the group ID of the
133 new process image file. The real user ID, real group ID, and supplementary group
134 IDs of the new process image remain the same as those of the calling process
135 image. If {_POSIX_SAVED_IDS} is defined, the effective user ID and effective
136 group ID of the new process image shall be saved (as the *saved set-user-ID* and the
137 *saved set-group-ID*) for use by the *setuid()* function.

138 The new process image also inherits the following attributes from the calling pro-
139 cess image:

- 140 (1) Process ID
- 141 (2) Parent process ID
- 142 (3) Process group ID
- 143 (4) Session membership
- 144 (5) Real user ID
- 145 (6) Real group ID

- 146 (7) Supplementary group IDs
- 147 (8) Time left until an alarm clock signal (see 3.4.1)
- 148 (9) Current working directory
- 149 (10) Root directory
- 150 (11) File mode creation mask (see 5.3.3)
- 151 (12) Process signal mask (see 3.3.5)
- 152 (13) Pending signals (see 3.3.6)
- 153 (14) *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* (see 4.5.2)

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

159 Upon successful completion, the *exec* functions shall mark for update the *st_atime*
 160 field of the file. If the *exec* function failed, but was able to locate the *process image*
 161 *file*, whether the *st_atime* field is marked for update is unspecified. Should the
 162 *exec* function succeed, the process image file shall be considered to have been
 163 *open()*-ed. The corresponding *close()* shall be considered to occur at a time after
 164 this open, but before process termination or successful completion of a subsequent
 165 call to one of the *exec* functions.

166 The *argv[]* and *envp[]* arrays of pointers and the strings to which those arrays
 167 point shall not be modified by a call to one of the *exec* functions, except as a conse-
 168 quence of replacing the process image.

169 3.1.2.3 Returns

170 If one of the *exec* functions returns to the calling process image, an error has
 171 occurred; the return value shall be -1, and *errno* shall be set to indicate the error.

172 3.1.2.4 Errors

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

175 [E2BIG] The number of bytes used by the argument list and the environ-
 176 ment list of the new process image is greater than the system-
 177 imposed limit of {ARG_MAX} bytes.

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

183 [ENAMETOOLONG]

184 The length of the *path* or *file* arguments, or an element of the

environment variable `PATH` prefixed to a file, exceeds `{PATH_MAX}`, or a pathname component is longer than `{NAME_MAX}` and `{_POSIX_NO_TRUNC}` is in effect for that file.

[ENOENT] One or more components of the pathname of the new process image file do not exist, or the *path* or *file* argument points to an empty string.

[ENOTDIR] A component of the path prefix of the new process image file is not a directory.

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

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

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

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

3.1.2.5 Cross-References

alarm(), 3.4.1; *chmod()*, 5.6.4; *_exit()*, 3.2.2; *fcntl()*, 6.5.2; *fork()*, 3.1.1; *setuid()*, 4.2.2; `<signal.h>`, 3.3.1; *sigprocmask()*, 3.3.5; *sigpending()*, 3.3.6; *stat()*, 5.6.2; `<sys/stat.h>`, 5.6.1; *times()*, 4.5.2; *umask()*, 5.3.3; 2.6.

3.2 Process Termination

There are two kinds of process termination:

(1) *Normal termination* occurs by a return from *main()* or when requested with the *exit()* or *_exit()* functions.

(2) *Abnormal termination* occurs when requested by the *abort()* function or when some signals are received (see 3.3.1).

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

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

218 **3.2.1 Wait for Process Termination**219 Functions: *wait()*, *waitpid()*220 **3.2.1.1 Synopsis**

```

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

```

225 **3.2.1.2 Description**

226 The *wait()* and *waitpid()* functions allow the calling process to obtain status infor-
 227 mation pertaining to one of its child processes. Various options permit status
 228 information to be obtained for child processes that have terminated or stopped. If
 229 status information is available for two or more child processes, the order in which
 230 their status is reported is unspecified.

231 The *wait()* function shall suspend execution of the calling process until status
 232 information for one of its terminated child processes is available, or until a signal
 233 whose action is either to execute a signal-catching function or to terminate the
 234 process is delivered. If status information is available prior to the call to *wait()*,
 235 return shall be immediate.

236 The *waitpid()* function shall behave identically to the *wait()* function if the *pid*
 237 argument has a value of -1 and the *options* argument has a value of zero. Other-
 238 wise, its behavior shall be modified by the values of the *pid* and *options*
 239 arguments.

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

- 243 (1) If *pid* is equal to -1 , status is requested for any child process. In this
 244 respect, *waitpid()* is then equivalent to *wait()*.
- 245 (2) If *pid* is greater than zero, it specifies the process ID of a single child pro-
 246 cess for which status is requested.
- 247 (3) If *pid* is equal to zero, status is requested for any child process whose
 248 process group ID is equal to that of the calling process.
- 249 (4) If *pid* is less than -1 , status is requested for any child process whose pro-
 250 cess group ID is equal to the absolute value of *pid*.

251 The *options* argument is constructed from the bitwise inclusive OR of zero or more
 252 of the following flags, defined in the header `<sys/wait.h>`:

- 253 **WNOHANG** The *waitpid()* function shall not suspend execution of the cal-
 254 ling process if status is not immediately available for one of the
 255 child processes specified by *pid*.
- 256 **WUNTRACED** If the implementation supports job control, the status of any
 257 child processes specified by *pid* that are stopped, and whose

258 status has not yet been reported since they stopped, shall also
259 be reported to the requesting process.

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

270 **WIFEXITED(*stat_val*)**
271 This macro evaluates to a nonzero value if status was returned
272 for a child process that terminated normally.

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

278 **WIFSIGNALED(*stat_val*)**
279 This macro evaluates to a nonzero value if status was returned
280 for a child process that terminated due to the receipt of a signal
281 that was not caught (see 3.3.1).

282 **WTERMSIG(*stat_val*)**
283 If the value of **WIFSIGNALED(*stat_val*)** is nonzero, this macro
284 evaluates to the number of the signal that caused the termina-
285 tion of the child process.

286 **WIFSTOPPED(*stat_val*)**
287 This macro evaluates to a nonzero value if status was returned
288 for a child process that is currently stopped.

289 **WSTOPSIG(*stat_val*)**
290 If the value of **WIFSTOPPED(*stat_val*)** is nonzero, this macro
291 evaluates to the number of the signal that caused the child pro-
292 cess to stop.

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

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

305

306 3.2.1.3 Returns

307 If the *wait()* or *waitpid()* functions return because the status of a child process is
 308 available, these functions shall return a value equal to the process ID of the child
 309 process for which status is reported. If the *wait()* or *waitpid()* functions return
 310 due to the delivery of a signal to the calling process, a value of -1 shall be
 311 returned and *errno* shall be set to [EINTR]. If the *waitpid()* function was invoked
 312 with WNOHANG set in *options*, has at least one child process specified by *pid* for
 313 which status is not available, and status is not available for any process specified
 314 by *pid*, a value of zero shall be returned. Otherwise, a value of -1 shall be
 315 returned, and *errno* shall be set to indicate the error.

316 3.2.1.4 Errors

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

319 [ECHILD] The calling process has no existing unwaited-for child
 320 processes.

321 [EINTR] The function was interrupted by a signal. The value of the
 322 location pointed to by *stat_loc* is undefined.

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

325 [ECHILD] The process or process group specified by *pid* does not exist or
 326 is not a child of the calling process.

327 [EINTR] The function was interrupted by a signal. The value of the
 328 location pointed to by *stat_loc* is undefined.

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

330 3.2.1.5 Cross-References

331 *_exit()*, 3.2.2; *fork()*, 3.1.1; *pause()*, 3.4.2; *times()*, 4.5.2; <signal.h>, 3.3.1.

332 3.2.2 Terminate a Process

333 Function: *_exit()*

334 **3.2.2.1 Synopsis**

335 `void _exit(int status);` |

336 **3.2.2.2 Description**

337 The `_exit()` function shall terminate the calling process with the following
338 consequences:

- 339 (1) All open file descriptors and directory streams in the calling process are
340 closed.
- 341 (2) If the parent process of the calling process is executing a `wait()` or `wait-`
342 `pid()`, it is notified of the termination of the calling process and the low
343 order 8 bits of *status* are made available to it; see 3.2.1.
- 344 (3) If the parent process of the calling process is not executing a `wait()` or
345 `waitpid()` function, the exit *status* code is saved for return to the parent
346 process whenever the parent process executes an appropriate subsequent
347 `wait()` or `waitpid()`.
- 348 (4) Termination of a process does not directly terminate its children. The
349 sending of a SIGHUP signal as described below indirectly terminates chil-
350 dren in some circumstances. Children of a terminated process shall be
351 assigned a new parent process ID, corresponding to an implementation-
352 defined system process.
- 353 (5) If the implementation supports the SIGCHLD signal, a SIGCHLD signal
354 shall be sent to the parent process.
- 355 (6) If the process is a controlling process, the SIGHUP signal shall be sent to
356 each process in the foreground process group of the controlling terminal
357 belonging to the calling process.
- 358 (7) If the process is a controlling process, the controlling terminal associated
359 with the session is disassociated from the session, allowing it to be
360 acquired by a new controlling process.
- 361 (8) If the implementation supports job control, and if the exit of the process
362 causes a process group to become orphaned, and if any member of the
363 newly orphaned process group is stopped, then a SIGHUP signal followed
364 by a SIGCONT signal shall be sent to each process in the newly orphaned
365 process group.

366 These consequences shall occur on process termination for any reason.

367 **3.2.2.3 Returns**

368 The `_exit()` function cannot return to its caller.

369 **3.2.2.4 Cross-References**

370 `close()`, 6.3.1; `sigaction()`, 3.3.4; `wait`, 3.2.1.

371 **3.3 Signals**372 **3.3.1 Signal Concepts**373 **3.3.1.1 Signal Names**

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

378	Symbolic	
379	Constant	Description
380	SIG_DFL	Request for default signal handling
381	SIG_IGN	Request that signal be ignored

382 The type *sigset_t* is used to represent sets of signals. It is always an integral or
 383 structure type. Several functions used to manipulate objects of type *sigset_t* are
 384 defined in 3.3.3.

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

390 The constants shown in Table 3-1 shall be supported by all implementations.

391 The constants shown in Table 3-2 shall be defined by all implementations. How-
 392 ever, implementations that do not support job control are not required to support
 393 these signals. If these signals are supported by the implementation, they shall
 394 behave in accordance with this part of ISO/IEC 9945. Otherwise, the implementa-
 395 tion shall not generate these signals, and attempts to send these signals or to
 396 examine or specify their actions shall return an error condition. See 3.3.2 and
 397 3.3.4.

398 **3.3.1.2 Signal Generation and Delivery**

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

404 Each process has an action to be taken in response to each signal defined by the
 405 system (see 3.3.1.3). A signal is said to be *delivered* to a process when the
 406 appropriate action for the process and signal is taken.

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

Table 3-1 – Required Signals

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

NOTE: The default actions are

- 1 Abnormal termination of the process.

Table 3-2 – Job Control Signals

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

NOTE: The default actions are

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

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

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

460 The determination of which action is taken in response to a signal is made at the
461 time the signal is delivered, allowing for any changes since the time of generation.
462 This determination is independent of the means by which the signal was origi-
463 nally generated. If a subsequent occurrence of a pending signal is generated, it is
464 implementation defined as to whether the signal is delivered more than once. The
465 order in which multiple, simultaneously pending signals are delivered to a pro-
466 cess is unspecified.

467 When any stop signal (SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU) is generated for a
468 process, any pending SIGCONT signals for that process shall be discarded. Con-
469 versely, when SIGCONT is generated for a process, all pending stop signals for
470 that process shall be discarded. When SIGCONT is generated for a process that is
471 stopped, the process shall be continued, even if the SIGCONT signal is blocked or
472 ignored. If SIGCONT is blocked and not ignored, it shall remain pending until it is
473 either unblocked or a stop signal is generated for the process.

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

476 3.3.1.3 Signal Actions

477 There are three types of actions that can be associated with a signal: SIG_DFL,
478 SIG_IGN, or a *pointer to a function*. Initially, all signals shall be set to SIG_DFL or
479 SIG_IGN prior to entry of the *main()* routine (see 3.1.2). The actions prescribed by
480 these values are as follows:

- 481 (1) SIG_DFL — signal-specific default action
 - 482 (a) The default actions for the signals defined in this part of
483 ISO/IEC 9945 are specified in Table 3-1 and Table 3-2.
 - 484 (b) If the default action is to stop the process, the execution of that pro-
485 cess is temporarily suspended. When a process stops, a SIGCHLD
486 signal shall be generated for its parent process, unless the parent
487 process has set the SA_NOCLDSTOP flag (see 3.3.4). While a process
488 is stopped, any additional signals that are sent to the process shall
489 not be delivered until the process is continued except SIGKILL,
490 which always terminates the receiving process. A process that is a
491 member of an orphaned process group shall not be allowed to stop
492 in response to the SIGTSTP, SIGTTIN, or SIGTTOU signals. In cases
493 where delivery of one of these signals would stop such a process, the
494 signal shall be discarded.

- 495 (c) Setting a signal action to SIG_DFL for a signal that is pending, and
496 whose default action is to ignore the signal (for example, SIGCHLD),
497 shall cause the pending signal to be discarded, whether or not it is
498 blocked.
- 499 (2) SIG_IGN — ignore signal
- 500 (a) Delivery of the signal shall have no effect on the process. The
501 behavior of a process is undefined after it ignores a SIGFPE, SIGILL,
502 or SIGSEGV signal that was not generated by the *kill()* function or
503 the *raise()* function defined by the C Standard [2].
- 504 (b) The system shall not allow the action for the signals SIGKILL or
505 SIGSTOP to be set to SIG_IGN.
- 506 (c) Setting a signal action to SIG_IGN for a signal that is pending shall
507 cause the pending signal to be discarded, whether or not it is
508 blocked.
- 509 (d) If a process sets the action for the SIGCHLD signal to SIG_IGN, the
510 behavior is unspecified.
- 511 (3) *pointer to a function* — catch signal
- 512 (a) On delivery of the signal, the receiving process is to execute the
513 signal-catching function at the specified address. After returning
514 from the signal-catching function, the receiving process shall
515 resume execution at the point at which it was interrupted.
- 516 (b) The signal-catching function shall be entered as a C language func-
517 tion call as follows:
- 518 `void func (int signo);` |
- 519 where *func* is the specified signal-catching function and *signo* is the
520 signal number of the signal being delivered.
- 521 (c) The behavior of a process is undefined after it returns normally
522 from a signal-catching function for a SIGFPE, SIGILL, or SIGSEGV
523 signal that was not generated by the *kill()* function or the *raise()*
524 function defined by the C Standard [2].
- 525 (d) The system shall not allow a process to catch the signals SIGKILL
526 and SIGSTOP.
- 527 (e) If a process establishes a signal-catching function for the SIGCHLD
528 signal while it has a terminated child process for which it has not
529 waited, it is unspecified whether a SIGCHLD signal is generated to
530 indicate that child process.
- 531 (f) When signal-catching functions are invoked asynchronously with
532 process execution, the behavior of some of the functions defined by
533 this part of ISO/IEC 9945 is unspecified if they are called from a
534 signal-catching function. The following table defines a set of func-
535 tions that shall be reentrant with respect to signals (that is, appli-
536 cations may invoke them, without restriction, from signal-catching
537 functions).

538	<code>_exit()</code>	<code>fstat()</code>	<code>read()</code>	<code>sysconf()</code>
539	<code>access()</code>	<code>getegid()</code>	<code>rename()</code>	<code>tcdrain()</code>
540	<code>alarm()</code>	<code>geteuid()</code>	<code>rmdir()</code>	<code>tcflow()</code>
541	<code>cfgetispeed()</code>	<code>getgid()</code>	<code>setgid()</code>	<code>tcflush()</code>
542	<code>cfgetospeed()</code>	<code>getgroups()</code>	<code>setpgid()</code>	<code>tcgetattr()</code>
543	<code>cfsetispeed()</code>	<code>getpgrp()</code>	<code>setsid()</code>	<code>tcgetpgrp()</code>
544	<code>cfsetospeed()</code>	<code>getpid()</code>	<code>setuid()</code>	<code>tcsendbreak()</code>
545	<code>chdir()</code>	<code>getppid()</code>	<code>sigaction()</code>	<code>tcsetattr()</code>
546	<code>chmod()</code>	<code>getuid()</code>	<code>sigaddset()</code>	<code>tcsetpgrp()</code>
547	<code>chown()</code>	<code>kill()</code>	<code>sigdelset()</code>	<code>time()</code>
548	<code>close()</code>	<code>link()</code>	<code>sigemptyset()</code>	<code>times()</code>
549	<code>creat()</code>	<code>lseek()</code>	<code>sigfillset()</code>	<code>umask()</code>
550	<code>dup2()</code>	<code>mkdir()</code>	<code>sigismember()</code>	<code>uname()</code>
551	<code>dup()</code>	<code>mkfifo()</code>	<code>sigpending()</code>	<code>unlink()</code>
552	<code>execle()</code>	<code>open()</code>	<code>sigprocmask()</code>	<code>utime()</code>
553	<code>execve()</code>	<code>pathconf()</code>	<code>sigsuspend()</code>	<code>wait()</code>
554	<code>fcntl()</code>	<code>pause()</code>	<code>sleep()</code>	<code>waitpid()</code>
555	<code>fork()</code>	<code>pipe()</code>	<code>stat()</code>	<code>write()</code>

556 All POSIX.1 functions not in the preceding table and all functions
557 defined in the C Standard [2] not stated to be callable from a
558 signal-catching function are considered to be *unsafe* with respect to
559 signals. In the presence of signals, all functions defined by this part
560 of ISO/IEC 9945 or by the C Standard [2] shall behave as defined (by
561 the defining standard) when called from or interrupted by a signal-
562 catching function, with a single exception: when a signal interrupts
563 an unsafe function and the signal-catching function calls an unsafe
564 function, the behavior is undefined.

555 3.3.1.4 Signal Effects on Other Functions

556 Signals affect the behavior of certain functions defined by this part of
557 ISO/IEC 9945 if delivered to a process while it is executing such a function. If the
558 action of the signal is to terminate the process, the process shall be terminated
559 and the function shall not return. If the action of the signal is to stop the process,
560 the process shall stop until continued or terminated. Generation of a SIGCONT
561 signal for the process causes the process to be continued, and the original function
562 shall continue at the point where the process was stopped. If the action of the sig-
563 nal is to invoke a signal-catching function, the signal-catching function shall be
564 invoked; in this case, the original function is said to be *interrupted* by the signal.
565 If the signal-catching function executes a `return`, the behavior of the interrupted
566 function shall be as described individually for that function. Signals that are
567 ignored shall not affect the behavior of any function; signals that are blocked shall
568 not affect the behavior of any function until they are delivered.

579 3.3.2 Send a Signal to a Process

580 Function: *kill()*

581 3.3.2.1 Synopsis

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

585 3.3.2.2 Description

586 The *kill()* function shall send a signal to a process or a group of processes
587 specified by *pid*. The signal to be sent is specified by *sig* and is either one from
588 the list given in 3.3.1.1 or zero. If *sig* is zero (the null signal), error checking is
589 performed, but no signal is actually sent. The null signal can be used to check the
590 validity of *pid*.

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

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

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

601 If *pid* is `-1`, the behavior of the *kill()* function is unspecified.

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

605 If the value of *pid* causes *sig* to be generated for the sending process, and if *sig* is
606 not blocked, either *sig* or at least one pending unblocked signal shall be delivered
607 to the sending process before the *kill()* function returns.

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

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

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

617 **3.3.2.3 Returns**

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

620 **3.3.2.4 Errors**

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

623	[EINVAL]	The value of the <i>sig</i> argument is an invalid or unsupported sig-
624		nal number.
625	[EPERM]	The process does not have permission to send the signal to any
626		receiving process.
627	[ESRCH]	No process or process group can be found corresponding to that
628		specified by <i>pid</i> .

629 **3.3.2.5 Cross-References**

630 *getpid()*, 4.1.1; *setsid()*, 4.3.2; *sigaction()*, 3.3.4; `<signal.h>`, 3.3.1.

631 **3.3 Manipulate Signal Sets**

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

633 **3.3.3.1 Synopsis**

```
634 #include <signal.h>
635 int sigemptyset(sigset_t *set);
636 int sigfillset(sigset_t *set);
637 int sigaddset(sigset_t *set, int signo);
638 int sigdelset(sigset_t *set, int signo);
639 int sigismember(const sigset_t *set, int signo);
```

640 **3.3.3.2 Description**

641 The *sigsetops* primitives manipulate sets of signals. They operate on data objects
642 addressable by the application, not on any set of signals known to the system,
643 such as the set blocked from delivery to a process or the set pending for a process
644 (see 3.3.1).

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

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

649 Applications shall call either *sigemptyset()* or *sigfillset()* at least once for each
650 object of type *sigset_t* prior to any other use of that object. If such an object is not

651 initialized in this way, but is nonetheless supplied as an argument to any of the
652 *sigaddset()*, *sigdelset()*, *sigismember()*, *sigaction()*, *sigprocmask()*, *sigpending()*, or
653 *sigsuspend()* functions, the results are undefined.

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

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

659 3.3.3.3 Returns

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

665 3.3.3.4 Errors

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

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

671 3.3.3.5 Cross-References

672 *sigaction()*, 3.3.4; <signal.h>, 3.3.1; *sigpending()*, 3.3.6; *sigprocmask()*, 3.3.5;
673 *sigsuspend()*, 3.3.7.

674 3.3.4 Examine and Change Signal Action

675 Function: *sigaction()*

676 3.3.4.1 Synopsis

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

680 3.3.4.2 Description

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

684 The structure *sigaction*, used to describe an action to be taken, is defined in the

685 header `<signal.h>` to include at least the following members:

686	Member	Member	Description
687	Type	Name	
688	<code>void (*)()</code>	<code>sa_handler</code>	SIG_DFL, SIG_IGN, or pointer to a function.
689	<code>sigset_t</code>	<code>sa_mask</code>	Additional set of signals to be blocked during
690			execution of signal-catching function.
691	<code>int</code>	<code>sa_flags</code>	Special flags to affect behavior of signal.

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

696 If the argument *act* is not NULL, it points to a structure specifying the action to
697 be associated with the specified signal. If the argument *oact* is not NULL, the
698 action previously associated with the signal is stored in the location pointed to by
699 the argument *oact*. If the argument *act* is NULL, signal handling is unchanged by
700 this function call; thus, the call can be used to enquire about the current handling
701 of a given signal. The *sa_handler* field of the *sigaction* structure identifies the
702 action to be associated with the specified signal. If the *sa_handler* field specifies a
703 signal-catching function, the *sa_mask* field identifies a set of signals that shall be
704 added to the signal mask of the process before the signal-catching function is
705 invoked. The SIGKILL and SIGSTOP signals shall not be added to the signal mask
706 using this mechanism; this restriction shall be enforced by the system without
707 causing an error to be indicated.

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

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

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

713 If *sig* is SIGCHLD and the SA_NOCLDSTOP flag is not set in *sa_flags*, and the
714 implementation supports the SIGCHLD signal, a SIGCHLD signal shall be gen-
715 erated for the calling process whenever any of its child processes stop. If *sig* is
716 SIGCHLD and the SA_NOCLDSTOP flag is set in *sa_flags*, the implementation
717 shall not generate a SIGCHLD signal in this way.

718 When a signal is caught by a signal-catching function installed by the *sigaction()*
719 function, a new signal mask is calculated and installed for the duration of the
720 signal-catching function [or until a call to either the *sigprocmask()* or *sig-*
721 *suspend()* function is made]. This mask is formed by taking the union of the
722 current signal mask and the value of the *sa_mask* for the signal being delivered,
723 and then including the signal being delivered. If and when the user's signal
724 handler returns normally, the original signal mask is restored.

725 Once an action is installed for a specific signal, it remains installed until another
726 action is explicitly requested [by another call to the *sigaction()* function] or until
727 one of the *exec* functions is called.

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

735 If the *sigaction()* function fails, no new signal handler is installed.

736 It is unspecified whether an attempt to set the action for a signal that cannot be
737 caught or ignored to SIG_DFL is ignored or causes an error to be returned with
738 *errno* set to [EINVAL].

739 3.3.4.3 Returns

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

742 3.3.4.4 Errors

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

745	[EINVAL]	The value of the <i>sig</i> argument is an invalid or unsupported sig-
746		nal number, or an attempt was made to catch a signal that can-
747		not be caught or to ignore a signal that cannot be ignored. See
748		3.3.1.1.

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

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

754 3.3.4.5 Cross-References

755 *kill()*, 3.3.2; <signal.h>, 3.3.1; *sigprocmask()*, 3.3.5; *sigsetops*, 3.3.3; *sig-*
756 *suspend()*, 3.3.7.

757 3.3.5 Examine and Change Blocked Signals

758 Function: *sigprocmask()*

759 3.3.5.1 Synopsis

760 #include <signal.h>

761 int sigprocmask(int *how*, const sigset_t **set*, sigset_t **oset*);

762 **3.3.5.2 Description**

763 The *sigprocmask()* function is used to examine or change (or both) the signal
764 mask of the calling process. If the value of the argument *set* is not NULL, it
765 points to a set of signals to be used to change the currently blocked set.

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

769	Name	Description
770	SIG_BLOCK	The resulting set shall be the union of the current set and
771		the signal set pointed to by the argument <i>set</i> .
772	SIG_UNBLOCK	The resulting set shall be the intersection of the current set
773		and the complement of the signal set pointed to by the argu-
774		ment <i>set</i> .
775	SIG_SETMASK	The resulting set shall be the signal set pointed to by the
776		argument <i>set</i> .

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

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

785 It is not possible to block the SIGKILL and SIGSTOP signals; this shall be enforced
786 by the system without causing an error to be indicated.

787 If any of the SIGFPE, SIGILL, or SIGSEGV signals are generated while they are
788 blocked, the result is undefined, unless the signal was generated by a call to the
789 *kill()* function or the *raise()* function defined by the C Standard [2].

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

792 **3.3.5.3 Returns**

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

795 **3.3.5.4 Errors**

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

798	[EINVAL]	The value of the <i>how</i> argument is not equal to one of the
799		defined values.

800 3.3.5.5 Cross-References

801 *sigaction()*, 3.3.4; `<signal.h>`, 3.3.1; *sigpending()*, 3.3.6; *sigsetops*, 3.3.3; *sig-*
802 *suspend()*, 3.3.7.

803 3.3.6 Examine Pending Signals

804 Function: *sigpending()*

805 3.3.6.1 Synopsis

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

808 3.3.6.2 Description

809 The *sigpending()* function shall store the set of signals that are blocked from
810 delivery and pending for the calling process in the space pointed to by the argu-
811 ment *set*.

812 3.3.6.3 Returns

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

815 3.3.6.4 Errors

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

819 3.3.6.5 Cross-References

820 `<signal.h>`, 3.3.1; *sigprocmask()*, 3.3.5; *sigsetops*, 3.3.3.

821 3.3.7 Wait for a Signal

822 Function: *sigsuspend()*

823 3.3.7.1 Synopsis

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

826 **3.3.7.2 Description**

827 The *sigsuspend()* function replaces the signal mask of the process with the set of
 828 signals pointed to by the argument *sigmask* and then suspends the process until
 829 delivery of a signal whose action is either to execute a signal-catching function or
 830 to terminate the process.

831 If the action is to terminate the process, the *sigsuspend()* function shall not
 832 return. If the action is to execute a signal-catching function, the *sigsuspend()*
 833 shall return after the signal-catching function returns, with the signal mask
 834 restored to the set that existed prior to the *sigsuspend()* call.

835 It is not possible to block those signals that cannot be ignored, as documented in
 836 3.3.1; this shall be enforced by the system without causing an error to be
 837 indicated.

838 **3.3.7.3 Returns**

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

842 **3.3.7.4 Errors**

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

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

847 **3.3.7.5 Cross-References**

848 *pause()*, 3.4.2; *sigaction()*, 3.3.4; *<signal.h>*, 3.3.1; *sigpending()*, 3.3.6; *sigproc-*
 849 *mask()*, 3.3.5; *sigsetops*, 3.3.3.

850 **3.4 Timer Operations**

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

855 **3.4.1 Schedule Alarm**

856 Function: *alarm()*

857 **3.4.1.1 Synopsis**

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

859 **3.4.1.2 Description**

860 The *alarm()* function shall cause the system to send the calling process a
861 SIGALRM signal after the number of real-time seconds specified by *seconds* have
862 elapsed.

863 Processor scheduling delays may cause the process actually not to begin handling
864 the signal until after the desired time.

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

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

869 **3.4.1.3 Returns**

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

874 **3.4.1.4 Errors**

875 The *alarm()* function is always successful, and no return value is reserved to indi-
876 cate an error.

877 **3.4.1.5 Cross-References**

878 *exec*, 3.1.2; *fork()*, 3.1.1; *pause()*, 3.4.2; *sigaction()*, 3.3.4; `<signal.h>`, 3.3.1.

879 **3.4.2 Suspend Process Execution**

880 Function: *pause()*

881 **3.4.2.1 Synopsis**

882 `int pause(void);` |

883 **3.4.2.2 Description**

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

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

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

889 **3.4.2.3 Returns**

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

893 **3.4.2.4 Errors**

894 If any of the following conditions occur, the *pause()* function shall return -1 and
 895 set *errno* to the corresponding value:

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

898 **3.4.2.5 Cross-References**

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

900 **3.4.3 Delay Process Execution**

901 Function: *sleep()*

902 **3.4.3.1 Synopsis**

903 unsigned int *sleep*(unsigned int *seconds*);

904 **3.4.3.2 Description**

905 The *sleep()* function shall cause the current process to be suspended from execu-
 906 tion until either the number of real-time seconds specified by the argument
 907 *seconds* have elapsed or a signal is delivered to the calling process and its action
 908 is to invoke a signal-catching function or to terminate the process. The suspen-
 909 sion time may be longer than requested due to the scheduling of other activity by
 910 the system.

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

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

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

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

931 **3.4.3.3 Returns**

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

936 **3.4.3.4 Errors**

937 The *sleep()* function is always successful, and no return value is reserved to indi-
938 cate an error.

939 **3.4.3.5 Cross-References**

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

Section 4: Process Environment

1 **4.1 Process Identification**

2 **4.1.1 Get Process and Parent Process IDs**

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

4 **4.1.1.1 Synopsis**

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

8 **4.1.1.2 Description**

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

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

11 **4.1.1.3 Returns**

12 See 4.1.1.2.

13 **4.1.1.4 Errors**

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

16 **4.1.1.5 Cross-References**

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

18 **4.2 User Identification**

19 **4.2.1 Get Real User, Effective User, Real Group, and Effective Group IDs**

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

21 **4.2.1.1 Synopsis**

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

27 **4.2.1.2 Description**

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

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

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

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

32 **4.2.1.3 Returns**

33 See 4.2.1.2.

34 **4.2.1.4 Errors**

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

37 **4.2.1.5 Cross-References**

38 *setuid()*, 4.2.2.

39 **4.2.2 Set User and Group IDs**

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

41 **4.2.2.1 Synopsis**

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

45 **4.2.2.2 Description**46 If `{_POSIX_SAVED_IDS}` is defined:

- 47 (1) If the process has appropriate privileges, the *setuid()* function sets the
48 real user ID, effective user ID, and the saved set-user-ID to *uid*.
- 49 (2) If the process does not have appropriate privileges, but *uid* is equal to the
50 real user ID or the saved set-user-ID, the *setuid()* function sets the effective
51 user ID to *uid*; the real user ID and saved set-user-ID remain
52 unchanged by this function call.
- 53 (3) If the process has appropriate privileges, the *setgid()* function sets the
54 real group ID, effective group ID, and the saved set-group-ID to *gid*.
- 55 (4) If the process does not have appropriate privileges, but *gid* is equal to the
56 real group ID or the saved set-group-ID, the *setgid()* function sets the
57 effective group ID to *gid*; the real group ID and saved set-group-ID remain
58 unchanged by this function call.

59 Otherwise:

- 60 (1) If the process has appropriate privileges, the *setuid()* function sets the
61 real user ID and effective user ID to *uid*.
- 62 (2) If the process does not have appropriate privileges, but *uid* is equal to the
63 real user ID, the *setuid()* function sets the effective user ID to *uid*; the
64 real user ID remains unchanged by this function call.
- 65 (3) If the process has appropriate privileges, the *setgid()* function sets the
66 real group ID and effective group ID to *gid*.
- 67 (4) If the process does not have appropriate privileges, but *gid* is equal to the
68 real group ID, the *setgid()* function sets the effective group ID to *gid*; the
69 real group ID remains unchanged by this function call.

70 Any supplementary group IDs of the calling process remain unchanged by these
71 function calls.72 **4.2.2.3 Returns**73 Upon successful completion, a value of zero is returned. Otherwise, a value of `-1`
74 is returned and *errno* is set to indicate the error.75 **4.2.2.4 Errors**76 If any of the following conditions occur, the *setuid()* function shall return `-1` and
77 set *errno* to the corresponding value:

- 78 [EINVAL] The value of the *uid* argument is invalid and not supported by
79 the implementation.
- 80 [EPERM] The process does not have appropriate privileges and *uid* does
81 not match the real user ID or, if `{_POSIX_SAVED_IDS}` is
82 defined, the saved set-user-ID.

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

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

90 4.2.2.5 Cross-References

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

92 4.2.3 Get Supplementary Group IDs

93 Function: *getgroups()*

94 4.2.3.1 Synopsis

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

97 4.2.3.2 Description

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

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

106 4.2.3.3 Returns

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

110 4.2.3.4 Errors

111 If any of the following conditions occur, the *getgroups()* function shall return *-1*
112 and set *errno* to the corresponding value:

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

115 **4.2.3.5 Cross-References**116 *setgid()*, 4.2.2.117 **4.2.4 Get User Name**118 Functions: *getlogin()* |119 **4.2.4.1 Synopsis**120 `char *getlogin(void);` |121 **4.2.4.2 Description**

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

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

128 |

129 **4.2.4.3 Returns**

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

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

134 |

135 **4.2.4.4 Errors**

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

139 **4.2.4.5 Cross-References**140 *getpwnam()*, 9.2.2; *getpwuid()*, 9.2.2.

141 **4.3 Process Groups**

142 **4.3.1 Get Process Group ID**

143 Function: *getpgrp()*

144 **4.3.1.1 Synopsis**

```
145 #include <sys/types.h>  
146 pid_t getpgrp(void);
```

147 **4.3.1.2 Description**

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

149 **4.3.1.3 Returns**

150 See 4.3.1.2.

151 **4.3.1.4 Errors**

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

154 **4.3.1.5 Cross-References**

155 *setpgid()*, 4.3.3; *setsid()*, 4.3.2; *sigaction()*, 3.3.4.

156 **4.3.2 Create Session and Set Process Group ID**

157 Function: *setsid()*

158 **4.3.2.1 Synopsis**

```
159 #include <sys/types.h>  
160 pid_t setsid(void);
```

161 **4.3.2.2 Description**

162 If the calling process is not a process group leader, the *setsid()* function shall
163 create a new session. The calling process shall be the session leader of this new
164 session, shall be the process group leader of a new process group, and shall have
165 no controlling terminal. The process group ID of the calling process shall be set
166 equal to the process ID of the calling process. The calling process shall be the only
167 process in the new process group and the only process in the new session.

168 **4.3.2.3 Returns**

169 Upon successful completion, the *setsid()* function returns the value of the process
 170 group ID of the calling process. Otherwise, a value of -1 is returned and *errno* is
 171 set to indicate the error.

172 **4.3.2.4 Errors**

173 If any of the following conditions occur, the *setsid()* function shall return -1 and
 174 set *errno* to the corresponding value:

175 [EPERM] The calling process is already a process group leader, or the
 176 process group ID of a process other than the calling process
 177 matches the process ID of the calling process.

178 **4.3.2.5 Cross-References**

179 *exec*, 3.1.2; *_exit()*, 3.2.2; *fork()*, 3.1.1; *getpid()*, 4.1.1; *kill()*, 3.3.2; *setpgid()*, 4.3.3;
 180 *sigaction()*, 3.3.4.

181 **4.3.3 Set Process Group ID for Job Control**

182 Function: *setpgid()*

183 **4.3.3.1 Synopsis**

184 #include <sys/types.h>
 185 int setpgid(pid_t *pid*, pid_t *pgid*);

186 **4.3.3.2 Description**

187 If $\{_POSIX_JOB_CONTROL\}$ is defined:

188 The *setpgid()* function is used to either join an existing process group or
 189 create a new process group within the session of the calling process. The
 190 process group ID of a session leader shall not change. Upon successful com-
 191 pletion, the process group ID of the process with a process ID that matches
 192 *pid* shall be set to *pgid*. As a special case, if *pid* is zero, the process ID of
 193 the calling process shall be used. Also, if *pgid* is zero, the process ID of the
 194 indicated process shall be used.

195 Otherwise:

196 Either the implementation shall support the *setpgid()* function as described
 197 above or the *setpgid()* function shall fail.

198 **4.3.3.3 Returns**

199 Upon successful completion, the *setpgid()* function returns a value of zero. Other-
 200 wise, a value of -1 is returned and *errno* is set to indicate the error.

201 4.3.3.4 Errors

202 If any of the following conditions occur, the *setpgid()* function shall return *-1* and
203 set *errno* to the corresponding value:

204	[EACCES]	The value of the <i>pid</i> argument matches the process ID of a child
205		process of the calling process, and the child process has success-
206		fully executed one of the <i>exec</i> functions.
207	[EINVAL]	The value of the <i>pgid</i> argument is less than zero or is not a
208		value supported by the implementation.
209	[ENOSYS]	The <i>setpgid()</i> function is not supported by this implementation.
210	[EPERM]	The process indicated by the <i>pid</i> argument is a session leader.
211		The value of the <i>pid</i> argument is valid, but matches the process
212		ID of a child process of the calling process, and the child process
213		is not in the same session as the calling process.
214		The value of the <i>pgid</i> argument does not match the process ID
215		of the process indicated by the <i>pid</i> argument, and there is no
216		process with a process group ID that matches the value of the
217		<i>pgid</i> argument in the same session as the calling process.
218	[ESRCH]	The value of the <i>pid</i> argument does not match the process ID of
219		the calling process or of a child process of the calling process.

220 4.3.3.5 Cross-References

221 *getpgrp()*, 4.3.1; *setsid()*, 4.3.2; *tcsetpgrp()*, 7.2.4; *exec*, 3.1.2.

222 4.4 System Identification

223 4.4.1 Get System Name

224 Function: *uname()*

225 4.4.1.1 Synopsis

```
226 #include <sys/utsname.h>
227 int uname(struct utsname *name);
```

228 4.4.1.2 Description

229 The *uname()* function stores information identifying the current operating system
230 in the structure pointed to by the argument *name*.

231 The structure *utsname* is defined in the header *<sys/utsname.h>* and contains
232 at least the members shown in Table 4-1.

Table 4-1 – *uname()* Structure Members

Member Name	Description
<i>sysname</i>	Name of this implementation of the operating system.
<i>nodename</i>	Name of this node within an implementation-specified communications network.
<i>release</i>	Current release level of this implementation.
<i>version</i>	Current version level of this release.
<i>machine</i>	Name of the hardware type on which the system is running.
Each of these data items is a null-terminated array of <i>char</i> .	
The format of each member is implementation defined. The system documentation (see 1.3.1.2) shall specify the source and format of each member and may specify the range of values for each member.	
The inclusion of the <i>nodename</i> member in this structure does not imply that it is sufficient information for interfacing to communications networks.	
4.4.1.3 Returns	
Upon successful completion, a nonnegative value is returned. Otherwise, a value of -1 is returned and <i>errno</i> is set to indicate the error.	
4.4.1.4 Errors	
This part of ISO/IEC 9945 does not specify any error conditions that are required to be detected for the <i>uname()</i> function. Some errors may be detected under conditions that are unspecified by this part of ISO/IEC 9945.	
4.5 Time	
4.5.1 Get System Time	
Function: <i>time()</i>	
4.5.1.1 Synopsis	
#include <time.h>	
time_t time(time_t *tloc);	

262 4.5.1.2 Description

263 The *time()* function returns the value of time in seconds since the Epoch.

264 The argument *tloc* points to an area where the return value is also stored. If *tloc*
265 is a NULL pointer, no value is stored.

266 4.5.1.3 Returns

267 Upon successful completion, *time()* returns the value of time. Otherwise, a value
268 of $((time_t) - 1)$ is returned and *errno* is set to indicate the error.

269 4.5.1.4 Errors

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

273 4.5.2 Get Process Times

274 Function: *times()*

275 4.5.2.1 Synopsis

276 #include <sys/times.h>
277 clock_t times(struct tms *buffer);

278 4.5.2.2 Description

279 The *times()* function shall fill the structure pointed to by *buffer* with time-
280 accounting information. The type *clock_t* and the *tms* structure are defined in
281 <sys/times.h>; the *tms* structure shall contain at least the following members:

282	Member	Member	
283	Type	Name	Description
284	<i>clock_t</i>	<i>tms_utime</i>	User CPU time.
285	<i>clock_t</i>	<i>tms_stime</i>	System CPU time.
286	<i>clock_t</i>	<i>tms_cutime</i>	User CPU time of terminated child processes.
287	<i>clock_t</i>	<i>tms_cstime</i>	System CPU time of terminated child processes.

288 All times are measured in terms of the number of clock ticks used.

289 The times of a terminated child process are included in the *tms_cutime* and
290 *tms_cstime* elements of the parent when a *wait()* or *waitpid()* function returns the
291 process ID of this terminated child. See 3.2.1. If a child process has not waited
292 for its terminated children, their times shall not be included in its times.

293 The value *tms_utime* is the CPU time charged for the execution of user
294 instructions.

295 The value *tms_time* is the CPU time charged for execution by the system on
296 behalf of the process.

297 The value *tms_cutime* is the sum of the *tms_utimes* and *tms_cutimes* of the child
298 processes.

299 The value *tms_cstime* is the sum of the *tms_stimes* and *tms_cstimes* of the child
300 processes.

301 4.5.2.3 Returns

302 Upon successful completion, *times()* shall return the elapsed real time, in clock |
303 ticks, since an arbitrary point in the past (for example, system start-up time).
304 This point does not change from one invocation of *times()* within the process to
305 another. The return value may overflow the possible range of type *clock_t*. If the
306 *times()* function fails, a value of $((\text{clock_t}) - 1)$ is returned and *errno* is set to indi-
307 cate the error.

308 4.5.2.4 Errors

309 This part of ISO/IEC 9945 does not specify any error conditions that are required |
310 to be detected for the *times()* function. Some errors may be detected under condi- |
311 tions that are unspecified by this part of ISO/IEC 9945.

312 4.5.2.5 Cross-References

313 *exec*, 3.1.2; *fork()*, 3.1.1; *sysconf()*, 4.8.1; *time()*, 4.5.1; *wait()*, 3.2.1. |

314 4.6 Environment Variables

315 4.6.1 Environment Access

316 Function: *getenv()*

317 4.6.1.1 Synopsis

318 #include <stdlib.h>
319 char *getenv(const char *name); |

320 4.6.1.2 Description

321 The *getenv()* function searches the environment list (see 2.6) for a string of the
322 form *name=value* and returns a pointer to *value* if such a string is present. If the
323 specified *name* cannot be found, a NULL pointer is returned.

324 4.6.1.3 Returns

325 Upon successful completion, the *getenv()* function returns a pointer to a string
326 containing the *value* for the specified *name*, or a NULL pointer if the specified
327 *name* cannot be found. The return value from *getenv()* may point to static data
328 and, therefore, may be overwritten by each call. Unsuccessful completion shall
329 result in the return of a NULL pointer.

330 4.6.1.4 Errors

331 This part of ISO/IEC 9945 does not specify any error conditions that are required
332 to be detected for the *getenv()* function. Some errors may be detected under condi-
333 tions that are unspecified by this part of ISO/IEC 9945. |

334 4.6.1.5 Cross-References

335 3.1.2; 2.6.

336 4.7 Terminal Identification

337 4.7.1 Generate Terminal Pathname

338 Function: *ctermid()*

339 4.7.1.1 Synopsis

340 #include <stdio.h>
341 char *ctermid(char *s); |

342 4.7.1.2 Description

343 The *ctermid()* function generates a string that, when used as a pathname, refers
344 to the current controlling terminal for the current process.

345 If the *ctermid()* function returns a pathname, access to the file is not guaranteed.

346 4.7.1.3 Returns

347 If *s* is a NULL pointer, the string is generated in an area that may be static (and,
348 therefore, may be overwritten by each call), the address of which is returned.
349 Otherwise, *s* is assumed to point to an array of *char* of at least L_ctermid bytes; |
350 the string is placed in this array and the value of *s* is returned. The symbolic con-
351 stant L_ctermid is defined in <stdio.h> and shall have a value greater than
352 zero.

353 The *ctermid()* function shall return an empty string if the pathname that would
354 refer to the controlling terminal cannot be determined or if the function is
355 unsuccessful.

356 4.7.1.4 Errors

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

360 4.7.1.5 Cross-References

361 *ttyname()*, 4.7.2.

362 4.7.2 Determine Terminal Device Name

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

364 4.7.2.1 Synopsis

365 `char *ttyname(int fildes);`

366 `int isatty(int fildes);`

367 4.7.2.2 Description

368 The *ttyname()* function returns a pointer to a string containing a null-terminated
369 pathname of the terminal associated with file descriptor *fildes*.

370 The return value of *ttyname()* may point to static data that is overwritten by each
371 call.

372 The *isatty()* function returns 1 if *fildes* is a valid file descriptor associated with a
373 terminal, zero otherwise.

374 4.7.2.3 Returns

375 The *ttyname()* function returns a NULL pointer if *fildes* is not a valid file descrip-
376 tor associated with a terminal or if the pathname cannot be determined.

377 4.7.2.4 Errors

378 This part of ISO/IEC 9945 does not specify any error conditions that are required
379 to be detected for the *ttyname()* or *isatty()* functions. Some errors may be
380 detected under conditions that are unspecified by this part of ISO/IEC 9945.

381 4.8 Configurable System Variables

382 4.8.1 Get Configurable System Variables

383 Function: *sysconf*()

384 4.8.1.1 Synopsis

385 #include <unistd.h>
386 long sysconf(int *name*);

387 4.8.1.2 Description

388 The *sysconf*() function provides a method for the application to determine the
389 current value of a configurable system limit or option (*variable*).

390 The *name* argument represents the system variable to be queried. The implemen-
391 tation shall support all of the variables listed in Table 4-2 and may support oth-
392 ers. The variables in Table 4-2 come from <limits.h> or <unistd.h> and the
393 symbolic constants, defined in <unistd.h>, that are the corresponding values
394 used for *name*.

395 **Table 4-2 – Configurable System Variables**

397	Variable	<i>name</i> Value
398	{ARG_MAX}	{_SC_ARG_MAX}
399	{CHILD_MAX}	{_SC_CHILD_MAX}
400	clock ticks/second	{_SC_CLK_TCK}
401	{NGROUPS_MAX}	{_SC_NGROUPS_MAX}
402	{OPEN_MAX}	{_SC_OPEN_MAX}
403	{STREAM_MAX}	{_SC_STREAM_MAX}
404	{TZNAME_MAX}	{_SC_TZNAME_MAX}
405	{_POSIX_JOB_CONTROL}	{_SC_JOB_CONTROL}
406	{_POSIX_SAVED_IDS}	{_SC_SAVED_IDS}
407	{_POSIX_VERSION}	{_SC_VERSION}
408		

409 4.8.1.3 Returns

410 If *name* is an invalid value, *sysconf*() shall return -1. If the variable correspond-
411 ing to *name* is associated with functionality that is not supported by the system,
412 *sysconf*() shall return -1 without changing the value of *errno*.

413 Otherwise, the *sysconf*() function returns the current variable value on the sys-
414 tem. The value returned shall not be more restrictive than the corresponding
415 value described to the application when it was compiled with the
416 implementation's <limits.h> or <unistd.h>. The value shall not change dur-
417 ing the lifetime of the calling process.

418 **4.8.1.4 Errors**

419 If any of the following conditions occur, the *sysconf()* function shall return -1 and
420 set *errno* to the corresponding value:

421 [EINVAL] The value of the *name* argument is invalid.

422 **4.8.1.5 Special Symbol (CLK_TCK)**

423 The special symbol {CLK_TCK} shall yield the same result as
424 *sysconf(_SC_CLK_TCK)*. It shall be defined in <time.h>. The symbol
425 {CLK_TCK} may be evaluated by the implementation at run time or may be a con-
426 stant. This special symbol is obsolescent.

THIS PAGE WAS
BLANK IN THE ORIGINAL

Section 5: Files and Directories

1 The functions in this section perform the operating system services dealing with
2 the creation and removal of files and directories and the detection and
3 modification of their characteristics. They also provide the primary methods a
4 process will use to gain access to files and directories for subsequent I/O opera-
5 tions (see Section 6).

6 5.1 Directories

7 5.1.1 Format of Directory Entries

8 The header `<dirent.h>` defines a structure and a defined type used by the *direc-*
9 *tory* routines.

10 The internal format of directories is unspecified. |

11 The *readdir()* function returns a pointer to an object of type *struct dirent* that
12 includes the member:

13	Member	Member	Description
14	Type	Name	
15	<i>char</i> []	<i>d_name</i>	Null-terminated filename

16 The array of *char d_name* is of unspecified size, but the number of bytes preceding |
17 the terminating null character shall not exceed `(NAME_MAX)`.

18 5.1.2 Directory Operations

19 Functions: *opendir()*, *readdir()*, *rewinddir()*, *closedir()*

20 5.1.2.1 Synopsis

```
21 #include <sys/types.h>
22 #include <dirent.h>
23 DIR *opendir(const char *dirname);
24 struct dirent *readdir(DIR *dirp);
25 void rewinddir(DIR *dirp);
26 int closedir(DIR *dirp);
```

27 5.1.2.2 Description

28 The type *DIR*, which is defined in the header `<dirent.h>`, represents a *directory*
29 *stream*, which is an ordered sequence of all the directory entries in a particular
30 directory. Directory entries represent files; files may be removed from a directory
31 or added to a directory asynchronously to the operations described in this sub-
32 clause (5.1.2). The type *DIR* may be implemented using a file descriptor. In that
33 case, applications will only be able to open up to a total of `{OPEN_MAX}` files and
34 directories; see 5.3.1. A successful call to any of the *exec* functions shall close any
35 directory streams that are open in the calling process.

36 The *opendir()* function opens a directory stream corresponding to the directory
37 named by the *dirname* argument. The directory stream is positioned at the first
38 entry.

39 The *readdir()* function returns a pointer to a structure representing the directory
40 entry at the current position in the directory stream to which *dirp* refers, and
41 positions the directory stream at the next entry. It returns a NULL pointer upon
42 reaching the end of the directory stream.

43 The *readdir()* function shall not return directory entries containing empty names.
44 It is unspecified whether entries are returned for dot or dot-dot.

45 The pointer returned by *readdir()* points to data that may be overwritten by
46 another call to *readdir()* on the same directory stream. This data shall not be
47 overwritten by another call to *readdir()* on a different directory stream.

48 The *readdir()* function may buffer several directory entries per actual read opera-
49 tion; the *readdir()* function shall mark for update the *st_atime* field of the direc-
50 tory each time the directory is actually read.

51 The *rewinddir()* function resets the position of the directory stream to which *dirp*
52 refers to the beginning of the directory. It also causes the directory stream to
53 refer to the current state of the corresponding directory, as a call to *opendir()*
54 would have done. It does not return a value.

55 If a file is removed from or added to the directory after the most recent call to
56 *opendir()* or *rewinddir()*, whether a subsequent call to *readdir()* returns an entry
57 for that file is unspecified.

58 The *closedir()* function closes the directory stream referred to by *dirp* and returns
59 a value of zero if successful. Otherwise, it returns `-1` indicating an error. Upon
60 return, the value of *dirp* may no longer point to an accessible object of type *DIR*.
61 If a file descriptor is used to implement type *DIR*, that file descriptor shall be
62 closed.

63 If the *dirp* argument passed to any of these functions does not refer to a currently
64 open directory stream, the effect is undefined.

65 The result of using a directory stream after one of the *exec* family of functions is
66 undefined. After a call to the *fork()* function, either the parent or the child (but
67 not both) may continue processing the directory stream using *readdir()* or *rewind-*
68 *dir()* or both. If both the parent and child processes use these functions, the
69 result is undefined. Either or both processes may use *closedir()*.

70 **5.1.2.3 Returns**

71 Upon successful completion, *opendir()* returns a pointer to an object of type *DIR*.
 72 Otherwise, a value of *NULL* is returned and *errno* is set to indicate the error.

73 Upon successful completion, *readdir()* returns a pointer to an object of type *struct*
 74 *dirent*. When an error is encountered, a value of *NULL* is returned and *errno* is
 75 set to indicate the error. When the end of the directory is encountered, a value of
 76 *NULL* is returned and *errno* is unchanged by this function call.

77 Upon successful completion, *closedir()* returns a value of zero. Otherwise, a value
 78 of *-1* is returned and *errno* is set to indicate the error.

79 **5.1.2.4 Errors**

80 If any of the following conditions occur, the *opendir()* function shall return a value
 81 of *NULL* and set *errno* to the corresponding value:

82 [EACCES] Search permission is denied for a component of the path prefix |
 83 of *dirname*, or read permission is denied for the directory itself. |

84 [ENAMETOOLONG] The length of the *dirname* argument exceeds {PATH_MAX}, or a
 85 pathname component is longer than {NAME_MAX} while
 86 {_POSIX_NO_TRUNC} is in effect.
 87

88 [ENOENT] The named directory does not exist, or *dirname* points to an |
 89 empty string. |

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

91 For each of the following conditions, when the condition is detected, the *opendir()*
 92 function shall return a value of *NULL* and set *errno* to the corresponding value:

93 [EMFILE] Too many file descriptors are currently open for the process.

94 [ENFILE] Too many file descriptors are currently open in the system.

95 For each of the following conditions, when the condition is detected, the *readdir()*
 96 function shall return a value of *NULL* and set *errno* to the corresponding value:

97 [EBADF] The *dirp* argument does not refer to an open directory stream.

98 For each of the following conditions, when the condition is detected, the *closedir()*
 99 function shall return *-1* and set *errno* to the corresponding value:

100 [EBADF] The *dirp* argument does not refer to an open directory stream.

101 **5.1.2.5 Cross-References**

102 <dirent.h>, 5.1.1.

103 **5.2 Working Directory**

104 **5.2.1 Change Current Working Directory**

105 Function: *chdir()*

106 **5.2.1.1 Synopsis**

107 `int chdir(const char *path);` |

108 **5.2.1.2 Description**

109 The *path* argument points to the pathname of a directory. The *chdir()* function
110 causes the named directory to become the current working directory, that is, the
111 starting point for path searches of pathnames not beginning with slash.

112 If the *chdir()* function fails, the current working directory shall remain
113 unchanged by this function call.

114 **5.2.1.3 Returns**

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

117 **5.2.1.4 Errors**

118 If any of the following conditions occur, the *chdir()* function shall return -1 and
119 set *errno* to the corresponding value:

120 [EACCES] Search permission is denied for any component of the path-
121 name.

122 [ENAMETOOLONG] The *path* argument exceeds {PATH_MAX} in length, or a path-
123 name component is longer than {NAME_MAX} while
124 {_POSIX_NO_TRUNC} is in effect.
125

126 [ENOTDIR] A component of the pathname is not a directory.

127 [ENOENT] The named directory does not exist or *path* is an empty string.

128 **5.2.1.5 Cross-References**

129 *getcwd()*, 5.2.2.

130 **5.2.2 Get Working Directory Pathname** |131 Function: *getcwd()*132 **5.2.2.1 Synopsis**133 `char *getcwd(char *buf, size_t size);` |134 **5.2.2.2 Description**

135 The *getcwd()* function copies an absolute pathname of the current working direc-
 136 tory to the array of *char* pointed to by the argument *buf* and returns a pointer to
 137 the result. The *size* argument is the size in bytes of the array of *char* pointed to
 138 by the *buf* argument. If *buf* is a NULL pointer, the behavior of *getcwd()* is
 139 undefined.

140 **5.2.2.3 Returns**

141 If successful, the *buf* argument is returned. A NULL pointer is returned if an
 142 error occurs and the variable *errno* is set to indicate the error. The contents of *buf*
 143 after an error are undefined.

144 **5.2.2.4 Errors**

145 If any of the following conditions occur, the *getcwd()* function shall return a value
 146 of NULL and set *errno* to the corresponding value:

147 [EINVAL] The *size* argument is zero. |

148 [ERANGE] The *size* argument is greater than zero but smaller than the
 149 length of the pathname plus 1.

150 For each of the following conditions, if the condition is detected, the *getcwd()* func-
 151 tion shall return a value of NULL and set *errno* to the corresponding value:

152 [EACCES] Read or search permission was denied for a component of the
 153 pathname.

154 **5.2.2.5 Cross-References**155 *chdir()*, 5.2.1.

156 5.3 General File Creation

157 5.3.1 Open a File

158 Function: *open()*

159 5.3.1.1 Synopsis

```
160 #include <sys/types.h>
161 #include <sys/stat.h>
162 #include <fcntl.h>
163 int open(const char *path, int oflag, ...);
```

164 5.3.1.2 Description

165 The *open()* function establishes the connection between a file and a file descriptor.
166 It creates an open file description that refers to a file and a file descriptor that
167 refers to that open file description. The file descriptor is used by other I/O func-
168 tions to refer to that file. The *path* argument points to a pathname naming a file.

169 The *open()* function shall return a file descriptor for the named file that is the
170 lowest file descriptor not currently open for that process. The open file description
171 is new, and therefore the file descriptor does not share it with any other process
172 in the system. The file offset shall be set to the beginning of the file. The
173 FD_CLOEXEC file descriptor flag associated with the new file descriptor shall be
174 cleared. The file status flags and file access modes of the open file description
175 shall be set according to the value of *oflag*. The value of *oflag* is the bitwise
176 inclusive OR of values from the following list. See 6.5.1 for the definitions of the
177 symbolic constants. Applications shall specify exactly one of the first three values
178 (file access modes) below in the value of *oflag*:

179	O_RDONLY	Open for reading only.
180	O_WRONLY	Open for writing only.
181	O_RDWR	Open for reading and writing. The result is undefined if this
182		flag is applied to a FIFO.

183 Any combination of the remaining flags may be specified in the value of *oflag*:

184	O_APPEND	If set, the file offset shall be set to the end of the file prior to
185		each write.
186	O_CREAT	This option requires a third argument, <i>mode</i> , which is of type
187		<i>mode_t</i> . If the file exists, this flag has no effect, except as noted
188		under O_EXCL, below. Otherwise, the file is created; the file's
189		user ID shall be set to the effective user ID of the process; the
190		file's group ID shall be set to the group ID of the directory in
191		which the file is being created or to the effective group ID of the
192		process. The file permission bits (see 5.6.1) shall be set to the
193		value of <i>mode</i> except those set in the file mode creation mask of
194		the process (see 5.3.3). When bits in <i>mode</i> other than the file
195		permission bits are set, the effect is unspecified. The <i>mode</i>

196 argument does not affect whether the file is opened for reading, |
 197 for writing, or for both.

198 **O_EXCL** If **O_EXCL** and **O_CREAT** are set, *open()* shall fail if the file
 199 exists. The check for the existence of the file and the creation of
 200 the file if it does not exist shall be atomic with respect to other
 201 processes executing *open()* naming the same filename in the
 202 same directory with **O_EXCL** and **O_CREAT** set. If **O_EXCL** is
 203 set and **O_CREAT** is not set, the result is undefined. |

204 **O_NOCTTY** If set, and *path* identifies a terminal device, the *open()* function
 205 shall not cause the terminal device to become the controlling
 206 terminal for the process (see 7.1.1.3).

207 **O_NONBLOCK**

208 (1) When opening a FIFO with **O_RDONLY** or **O_WRONLY** set:

209 (a) If **O_NONBLOCK** is set:
 210 An *open()* for reading-only shall return without
 211 delay. An *open()* for writing-only shall return an
 212 error if no process currently has the file open for
 213 reading.

214 (b) If **O_NONBLOCK** is clear:
 215 An *open()* for reading-only shall block until a process
 216 opens the file for writing. An *open()* for writing-only
 217 shall block until a process opens the file for reading.

218 (2) When opening a block special or character special file that
 219 supports nonblocking opens:

220 (a) If **O_NONBLOCK** is set:
 221 The *open()* shall return without waiting for the de-
 222 vice to be ready or available. Subsequent behavior of
 223 the device is device-specific.

224 (b) If **O_NONBLOCK** is clear:
 225 The *open()* shall wait until the device is ready or
 226 available before returning.

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

228 **O_TRUNC** If the file exists and is a regular file, and the file is successfully
 229 opened **O_RDWR** or **O_WRONLY**, it shall be truncated to zero
 230 length and the mode and owner shall be unchanged by this
 231 function call. **O_TRUNC** shall have no effect on FIFO special
 232 files or terminal device files. Its effect on other file types is |
 233 implementation defined. The result of using **O_TRUNC** with
 234 **O_RDONLY** is undefined.

235 If **O_CREAT** is set and the file did not previously exist, upon successful completion
 236 the *open()* function shall mark for update the *st_atime*, *st_ctime*, and *st_mtime*
 237 fields of the file and the *st_ctime* and *st_mtime* fields of the parent directory.

238 If `O_TRUNC` is set and the file did previously exist, upon successful completion
239 the `open()` function shall mark for update the `st_ctime` and `st_mtime` fields of the
240 file.

241 5.3.1.3 Returns

242 Upon successful completion, the function shall open the file and return a nonne-
243 gative integer representing the lowest numbered unused file descriptor. Other-
244 wise, it shall return `-1` and shall set `errno` to indicate the error. No files shall be
245 created or modified if the function returns `-1`.

246 5.3.1.4 Errors

247 If any of the following conditions occur, the `open()` function shall return `-1` and set
248 `errno` to the corresponding value:

- | | | |
|-----|----------------|---|
| 249 | [EACCES] | Search permission is denied on a component of the path prefix, |
| 250 | | or the file exists and the permissions specified by <i>oflag</i> are |
| 251 | | denied, or the file does not exist and write permission is denied |
| 252 | | for the parent directory of the file to be created, or <code>O_TRUNC</code> is |
| 253 | | specified and write permission is denied. |
| 254 | [EEXIST] | <code>O_CREAT</code> and <code>O_EXCL</code> are set and the named file exists. |
| 255 | [EINTR] | The <code>open()</code> operation was interrupted by a signal. |
| 256 | [EISDIR] | The named file is a directory, and the <i>oflag</i> argument specifies |
| 257 | | write or read/write access. |
| 258 | [EMFILE] | Too many file descriptors are currently in use by this process. |
| 259 | [ENAMETOOLONG] | |
| 260 | | The length of the <i>path</i> string exceeds <code>{PATH_MAX}</code> , or a path- |
| 261 | | name component is longer than <code>{NAME_MAX}</code> while |
| 262 | | <code>{_POSIX_NO_TRUNC}</code> is in effect. |
| 263 | [ENFILE] | Too many files are currently open in the system. |
| 264 | [ENOENT] | <code>O_CREAT</code> is not set and the named file does not exist, or |
| 265 | | <code>O_CREAT</code> is set and either the path prefix does not exist or the |
| 266 | | <i>path</i> argument points to an empty string. |
| 267 | [ENOSPC] | The directory or file system that would contain the new file can- |
| 268 | | not be extended. |
| 269 | [ENOTDIR] | A component of the path prefix is not a directory. |
| 270 | [ENXIO] | <code>O_NONBLOCK</code> is set, the named file is a FIFO, <code>O_WRONLY</code> is |
| 271 | | set, and no process has the file open for reading. |
| 272 | [EROFS] | The named file resides on a read-only file system and either |
| 273 | | <code>O_WRONLY</code> , <code>O_RDWR</code> , <code>O_CREAT</code> (if the file does not exist), or |
| 274 | | <code>O_TRUNC</code> is set in the <i>oflag</i> argument. |

275 **5.3.1.5 Cross-References**

276 *close()*, 6.3.1; *creat()*, 5.3.2; *dup()*, 6.2.1; *exec*, 3.1.2; *fcntl()*, 6.5.2; <fcntl.h>,
 277 6.5.1; *lseek()*, 6.5.3; *read()*, 6.4.1; <signal.h>, 3.3.1; *stat()*, 5.6.2;
 278 <sys/stat.h>, 5.6.1; *write()*, 6.4.2; *umask()*, 5.3.3; 3.3.1.4.

279 **5.3.2 Create a New File or Rewrite an Existing One**

280 Function: *creat()*

281 **5.3.2.1 Synopsis**

282 #include <sys/types.h>
 283 #include <sys/stat.h>
 284 #include <fcntl.h>
 285 int creat(const char *path, mode_t mode);

286 **5.3.2.2 Description**

287 The function call:

288 *creat(path, mode);*

289 is equivalent to:

290 *open(path, O_WRONLY | O_CREAT | O_TRUNC, mode);*

291 **5.3.2.3 Cross-References**

292 *open()*, 5.3.1; <sys/stat.h>, 5.6.1.

293 **5.3.3 Set File Creation Mask**

294 Function: *umask()*

295 **5.3.3.1 Synopsis**

296 #include <sys/types.h>
 297 #include <sys/stat.h>
 298 mode_t umask(mode_t cmask);

299 **5.3.3.2 Description**

300 The *umask()* routine sets the file mode creation mask of the process to *cmask* and
 301 returns the previous value of the mask. Only the file permission bits (see 5.6.1) of
 302 *cmask* are used; the meaning of the other bits is implementation defined.

303 The file mode creation mask of the process is used during *open()*, *creat()*, *mkdir()*,
 304 and *mkfifo()* calls to turn off permission bits in the *mode* argument supplied. Bit
 305 positions that are set in *cmask* are cleared in the mode of the created file.

306 5.3.3.3 Returns

307 The file permission bits in the value returned by *umask()* shall be the previous
308 value of the file mode creation mask. The state of any other bits in that value is
309 unspecified, except that a subsequent call to *umask()* with that returned value as
310 *cmask* shall leave the state of the mask the same as its state before the first call,
311 including any unspecified (by this part of ISO/IEC 9945) use of those bits.

312 5.3.3.4 Errors

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

315 5.3.3.5 Cross-References

316 *chmod()*, 5.6.4; *creat()*, 5.3.2; *mkdir()*, 5.4.1; *mkfifo()*, 5.4.2; *open()*, 5.3.1;
317 *<sys/stat.h>*, 5.6.1.

318 5.3.4 Link to a File

319 Function: *link()*

320 5.3.4.1 Synopsis

321 `int link(const char *existing, const char *new);`

322 5.3.4.2 Description

323 The argument *existing* points to a pathname naming an existing file. The argu-
324 ment *new* points to a pathname naming the new directory entry to be created.
325 Implementations may support linking of files across file systems. The *link()* func-
326 tion shall atomically create a new link for the existing file and increment the link
327 count of the file by one.

328 If the *link()* function fails, no link shall be created, and the link count of the file
329 shall remain unchanged by this function call.

330 The *existing* argument shall not name a directory unless the user has appropriate
331 privileges and the implementation supports using *link()* on directories.

332 The implementation may require that the calling process has permission to access
333 the existing file.

334 Upon successful completion, the *link()* function shall mark for update the *st_ctime*
335 field of the file. Also, the *st_ctime* and *st_mtime* fields of the directory that con-
336 tains the new entry are marked for update.

337 **5.3.4.3 Returns**

338 Upon successful completion, *link()* shall return a value of zero. Otherwise, a
 339 value of -1 is returned and *errno* is set to indicate the error.

340 **5.3.4.4 Errors**

341 If any of the following conditions occur, the *link()* function shall return -1 and set
 342 *errno* to the corresponding value:

- | | | | |
|---------------------------------|----------------|--|--|
| 343
344
345
346
347 | [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. | |
| 348 | [EEXIST] | The link named by <i>new</i> exists. | |
| 349
350 | [EMLINK] | The number of links to the file named by <i>existing</i> would exceed {LINK_MAX}. | |
| 351
352
353
354 | [ENAMETOOLONG] | The length of the <i>existing</i> or <i>new</i> string exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect. | |
| 355
356
357 | [ENOENT] | A component of either path prefix does not exist, the file named by <i>existing</i> does not exist, or either <i>existing</i> or <i>new</i> points to an empty string. | |
| 358 | [ENOSPC] | The directory that would contain the link cannot be extended. | |
| 359 | [ENOTDIR] | A component of either path prefix is not a directory. | |
| 360
361
362 | [EPERM] | The file named by <i>existing</i> is a directory, and either the calling process does not have appropriate privileges or the implementation prohibits using <i>link()</i> on directories. | |
| 363
364 | [EROFS] | The requested link requires writing in a directory on a read-only file system. | |
| 365
366
367 | [EXDEV] | The link named by <i>new</i> and the file named by <i>existing</i> are on different file systems, and the implementation does not support links between file systems. | |

368 **5.3.4.5 Cross-References**

369 *rename()*, 5.5.3; *unlink()*, 5.5.1.

370 5.4 Special File Creation

371 5.4.1 Make a Directory

372 Function: *mkdir()*

373 5.4.1.1 Synopsis

```
374 #include <sys/types.h>
375 #include <sys/stat.h>
376 int mkdir(const char *path, mode_t mode);
```

377 5.4.1.2 Description

378 The *mkdir()* routine creates a new directory with name *path*. The file permission
379 bits of the new directory are initialized from *mode*. The file permission bits of the
380 *mode* argument are modified by the file creation mask of the process (see 5.3.3).
381 When bits in *mode* other than the file permission bits are set, the meaning of
382 these additional bits is implementation defined.

383 The owner ID of the directory is set to the effective user ID of the process. The
384 directory's group ID shall be set to the group ID of the directory in which the direc-
385 tory is being created or to the effective group ID of the process.

386 The newly created directory shall be an empty directory.

387 Upon successful completion, the *mkdir()* function shall mark for update the
388 *st_atime*, *st_ctime*, and *st_mtime* fields of the directory. Also, the *st_ctime* and
389 *st_mtime* fields of the directory that contains the new entry are marked for
390 update.

391 5.4.1.3 Returns

392 A return value of zero indicates success. A return value of -1 indicates that an
393 error has occurred, and an error code is stored in *errno*. No directory shall be
394 created if the return value is -1.

395 5.4.1.4 Errors

396 If any of the following conditions occur, the *mkdir()* function shall return -1 and
397 set *errno* to the corresponding value:

398	[EACCES]	Search permission is denied on a component of the path prefix,
399		or write permission is denied on the parent directory of the
400		directory to be created.
401	[EEXIST]	The named file exists.
402	[EMLINK]	The link count of the parent directory would exceed
403		{LINK_MAX}.

404	[ENAMETOOLONG]	
405		The length of the <i>path</i> argument exceeds {PATH_MAX}, or a
406		pathname component is longer than {NAME_MAX} while
407		{_POSIX_NO_TRUNC} is in effect.
408	[ENOENT]	A component of the path prefix does not exist, or the <i>path</i> argu-
409		ment points to an empty string.
410	[ENOSPC]	The file system does not contain enough space to hold the con-
411		tents of the new directory or to extend the parent directory of
412		the new directory.
413	[ENOTDIR]	A component of the path prefix is not a directory.
414	[EROFS]	The parent directory of the directory being created resides on a
415		read-only file system.

416 5.4.1.5 Cross-References

417 *chmod()*, 5.6.4; *stat()*, 5.6.2; <sys/stat.h>, 5.6.1; *umask()*, 5.3.3.

418 5.4.2 Make a FIFO Special File

419 Function: *mkfifo()*

420 5.4.2.1 Synopsis

```
421 #include <sys/types.h>
422 #include <sys/stat.h>
423 int mkfifo(const char *path, mode_t mode);
```

424 5.4.2.2 Description

425 The *mkfifo()* routine creates a new FIFO special file named by the pathname
 426 pointed to by *path*. The file permission bits of the new FIFO are initialized from
 427 *mode*. The file permission bits of the *mode* argument are modified by the file crea-
 428 tion mask of the process (see 5.3.3). When bits in *mode* other than the file permis-
 429 sion bits are set, the effect is implementation defined.

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

433 Upon successful completion, the *mkfifo()* function shall mark for update the
 434 *st_atime*, *st_ctime*, and *st_mtime* fields of the file. Also, the *st_ctime* and *st_mtime*
 435 fields of the directory that contains the new entry are marked for update.

436 5.4.2.3 Returns

437 Upon successful completion, a value of zero is returned. Otherwise, a value of -1
 438 is returned, no FIFO is created, and *errno* is set to indicate the error.

439 5.4.2.4 Errors

440 If any of the following conditions occur, the *mkfifo()* function shall return -1 and
441 set *errno* to the corresponding value:

- | | | | |
|-----|----------------|---|--|
| 442 | [EACCES] | Search permission is denied on a component of the path prefix, | |
| 443 | | or write permission is denied on the parent directory of the file | |
| 444 | | to be created. | |
| 445 | [EEXIST] | The named file already exists. | |
| 446 | [ENAMETOOLONG] | | |
| 447 | | The length of the <i>path</i> string exceeds {PATH_MAX}, or a path- | |
| 448 | | name component is longer than {NAME_MAX} while | |
| 449 | | {_POSIX_NO_TRUNC} is in effect. | |
| 450 | [ENOENT] | A component of the path prefix does not exist, or the <i>path</i> argu- | |
| 451 | | ment points to an empty string. | |
| 452 | [ENOSPC] | The directory that would contain the new file cannot be | |
| 453 | | extended, or the file system is out of file allocation resources. | |
| 454 | [ENOTDIR] | A component of the path prefix is not a directory. | |
| 455 | [EROFS] | The named file resides on a read-only file system. | |

456 5.4.2.5 Cross-References

457 *chmod()*, 5.6.4; *exec*, 3.1.2; *pipe()*, 6.1.1; *stat()*, 5.6.2; <sys/stat.h>, 5.6.1;
458 *umask()*, 5.3.3.

459 5.5 File Removal

460 5.5.1 Remove Directory Entries

461 Function: *unlink()*

462 5.5.1.1 Synopsis

463 `int unlink(const char *path);`

464 5.5.1.2 Description

465 The *unlink()* function shall remove the link named by the pathname pointed to by
466 *path* and decrement the link count of the file referenced by the link.

467 When the link count of the file becomes zero and no process has the file open, the
468 space occupied by the file shall be freed and the file shall no longer be accessible.
469 If one or more processes have the file open when the last link is removed, the link
470 shall be removed before *unlink()* returns, but the removal of the file contents shall
471 be postponed until all references to the file have been closed.

472 The *path* argument shall not name a directory unless the process has appropriate
 473 privileges and the implementation supports using *unlink()* on directories. Appli-
 474 cations should use *rmdir()* to remove a directory.

475 Upon successful completion, the *unlink()* function shall mark for update the
 476 *st_ctime* and *st_mtime* fields of the parent directory. Also, if the link count of the
 477 file is not zero, the *st_ctime* field of the file shall be marked for update.

478 5.5.1.3 Returns

479 Upon successful completion, a value of zero shall be returned. Otherwise, a value
 480 of -1 shall be returned and *errno* shall be set to indicate the error. If -1 is
 481 returned, the named file shall not be changed by this function call.

482 5.5.1.4 Errors

483 If any of the following conditions occur, the *unlink()* function shall return -1 and
 484 set *errno* to the corresponding value:

485 [EACCES] Search permission is denied for a component of the path prefix,
 486 or write permission is denied on the directory containing the
 487 link to be removed.

488 [EBUSY] The directory named by the *path* argument cannot be unlinked
 489 because it is being used by the system or another process and
 490 the implementation considers this to be an error.

491 [ENAMETOOLONG] The length of the *path* argument exceeds {PATH_MAX}, or a
 492 pathname component is longer than {NAME_MAX} while
 493 {_POSIX_NO_TRUNC} is in effect.
 494

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

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

498 [EPERM] The file named by *path* is a directory, and either the calling
 499 process does not have appropriate privileges or the implemen-
 500 tation prohibits using *unlink()* on directories.

501 [EROFS] The directory entry to be unlinked resides on a read-only file
 502 system.

503 5.5.1.5 Cross-References

504 *close()*, 6.3.1; *link()*, 5.3.4; *open()*, 5.3.1; *rename()*, 5.5.3; *rmdir()*, 5.5.2.

505 5.5.2 Remove a Directory

506 Function: *rmdir()*

507 5.5.2.1 Synopsis

508 `int rmdir(const char *path);` |

509 5.5.2.2 Description

510 The *rmdir()* function removes a directory whose name is given by *path*. The
511 directory shall be removed only if it is an empty directory.

512 If the named directory is the root directory or the current working directory of any
513 process, it is unspecified whether the function succeeds or whether it fails and
514 sets *errno* to [EBUSY]. |

515 If the link count of the directory becomes zero and no process has the directory
516 open, the space occupied by the directory shall be freed and the directory shall no
517 longer be accessible. If one or more processes have the directory open when the
518 last link is removed, the dot and dot-dot entries, if present, are removed before
519 *rmdir()* returns and no new entries may be created in the directory, but the direc-
520 tory is not removed until all references to the directory have been closed.

521 Upon successful completion, the *rmdir()* function shall mark for update the
522 *st_ctime* and *st_mtime* fields of the parent directory.

523 5.5.2.3 Returns

524 Upon successful completion, a value of zero shall be returned. Otherwise, a value
525 of `-1` shall be returned and *errno* shall be set to indicate the error. If `-1` is
526 returned, the named directory shall not be changed by this function call. |

527 5.5.2.4 Errors

528 If any of the following conditions occur, the *rmdir()* function shall return `-1` and
529 set *errno* to the corresponding value:

530 [EACCES] Search permission is denied on a component of the path prefix, |
531 or write permission is denied on the parent directory of the |
532 directory to be removed. |

533 [EBUSY] The directory named by the *path* argument cannot be removed
534 because it is being used by another process and the implemen-
535 tation considers this to be an error.

536 [EEXIST] or [ENOTEMPTY]
537 The *path* argument names a directory that is not an empty
538 directory.

539 [ENAMETOOLONG]
540 The length of the *path* argument exceeds {PATH_MAX}, or a
541 pathname component is longer than {NAME_MAX} while

542 { _POSIX_NO_TRUNC } is in effect.

543 [ENOENT] The *path* argument names a nonexistent directory or points to
544 an empty string.

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

546 [EROFS] The directory entry to be removed resides on a read-only file
547 system.

548 **5.5.2.5 Cross-References**

549 *mkdir()*, 5.4.1; *unlink()*, 5.5.1.

550 **5.5.3 Rename a File**

551 Function: *rename()*

552 **5.5.3.1 Synopsis**

553 int rename(const char *old, const char *new);

554 **5.5.3.2 Description**

555 The *rename()* function changes the name of a file. The *old* argument points to the
556 pathname of the file to be renamed. The *new* argument points to the new path-
557 name of the file.

558 If the *old* argument and the *new* argument both refer to links to the same existing
559 file, the *rename()* function shall return successfully and perform no other action.

560 If the *old* argument points to the pathname of a file that is not a directory, the
561 *new* argument shall not point to the pathname of a directory. If the link named
562 by the *new* argument exists, it shall be removed and *old* renamed to *new*. In this
563 case, a link named *new* shall exist throughout the renaming operation and shall
564 refer either to the file referred to by *new* or *old* before the operation began. Write
565 access permission is required for both the directory containing *old* and the direc-
566 tory containing *new*.

567 If the *old* argument points to the pathname of a directory, the *new* argument shall
568 not point to the pathname of a file that is not a directory. If the directory named
569 by the *new* argument exists, it shall be removed and *old* renamed to *new*. In this
570 case, a link named *new* shall exist throughout the renaming operation and shall
571 refer either to the file referred to by *new* or *old* before the operation began. Thus,
572 if *new* names an existing directory, it shall be required to be an empty directory.

573 The *new* pathname shall not contain a path prefix that names *old*. Write access
574 permission is required for the directory containing *old* and the directory contain-
575 ing *new*. If the *old* argument points to the pathname of a directory, write access
576 permission may be required for the directory named by *old*, and, if it exists, the
577 directory named by *new*.

578 If the link named by the *new* argument exists and the link count of the file
579 becomes zero when it is removed and no process has the file open, the space occu-
580 pied by the file shall be freed and the file shall no longer be accessible. If one or
581 more processes have the file open when the last link is removed, the link shall be
582 removed before *rename()* returns, but the removal of the file contents shall be
583 postponed until all references to the file have been closed.

584 Upon successful completion, the *rename()* function shall mark for update the
585 *st_ctime* and *st_mtime* fields of the parent directory of each file.

586 5.5.3.3 Returns

587 Upon successful completion, a value of zero shall be returned. Otherwise, a value
588 of *-1* shall be returned and *errno* shall be set to indicate the error. If *-1* is
589 returned, neither the file named by *old* nor the file named by *new*, if either exists,
590 shall be changed by this function call.

591 5.5.3.4 Errors

592 If any of the following conditions occur, the *rename()* function shall return *-1* and
593 set *errno* to the corresponding value:

- | | | |
|-----|-------------------------|--|
| 594 | [EACCES] | A component of either path prefix denies search permission, or |
| 595 | | one of the directories containing <i>old</i> or <i>new</i> denies write per- |
| 596 | | missions, or write permission is required and is denied for a |
| 597 | | directory pointed to by the <i>old</i> or <i>new</i> arguments. |
| 598 | [EBUSY] | The directory named by <i>old</i> or <i>new</i> cannot be renamed because |
| 599 | | it is being used by the system or another process and the imple- |
| 600 | | mentation considers this to be an error. |
| 601 | [EEXIST] or [ENOTEMPTY] | |
| 602 | | The link named by <i>new</i> is a directory containing entries other |
| 603 | | than dot and dot-dot. |
| 604 | [EINVAL] | The <i>new</i> directory pathname contains a path prefix that names |
| 605 | | the <i>old</i> directory. |
| 606 | [EISDIR] | The <i>new</i> argument points to a directory, and the <i>old</i> argument |
| 607 | | points to a file that is not a directory. |
| 608 | [ENAMETOOLONG] | |
| 609 | | The length of the <i>old</i> or <i>new</i> argument exceeds {PATH_MAX}, or |
| 610 | | a pathname component is longer than {NAME_MAX} while |
| 611 | | {_POSIX_NO_TRUNC} is in effect. |
| 612 | [EMLINK] | The file named by <i>old</i> is a directory, and the link count of the |
| 613 | | parent directory of <i>new</i> would exceed {LINK_MAX}. |
| 614 | [ENOENT] | The link named by the <i>old</i> argument does not exist, or either |
| 615 | | <i>old</i> or <i>new</i> points to an empty string. |
| 616 | [ENOSPC] | The directory that would contain <i>new</i> cannot be extended. |

- 617 [ENOTDIR] A component of either path prefix is not a directory, or the *old*
618 argument names a directory and the *new* argument names a
619 nondirectory file.
- 620 [EROFS] The requested operation requires writing in a directory on a
621 read-only file system.
- 622 [EXDEV] The links named by *new* and *old* are on different file systems,
623 and the implementation does not support links between file
624 systems.

625 **5.5.3.5 Cross-References**

626 *link()*, 5.3.4; *rmdir()*, 5.5.2; *unlink()*, 5.5.1.

627 **5.6 File Characteristics**

628 **5.6.1 File Characteristics: Header and Data Structure**

629 The header `<sys/stat.h>` defines the structure *stat*, which includes the
630 members shown in Table 5-1, returned by the functions *stat()* and *fstat()*.

631 **Table 5-1 – *stat* Structure**

632	Member	Member	
633	Type	Name	Description
634			
635	<i>mode_t</i>	<i>st_mode</i>	File mode (see 5.6.1.2).
636	<i>ino_t</i>	<i>st_ino</i>	File serial number.
637	<i>dev_t</i>	<i>st_dev</i>	ID of device containing this file.
638	<i>nlink_t</i>	<i>st_nlink</i>	Number of links.
639	<i>uid_t</i>	<i>st_uid</i>	User ID of the owner of the file.
640	<i>gid_t</i>	<i>st_gid</i>	Group ID of the group of the file.
641	<i>off_t</i>	<i>st_size</i>	For regular files, the file size in bytes. For other file types, the use of
642			this field is unspecified.
643	<i>time_t</i>	<i>st_atime</i>	Time of last access.
644	<i>time_t</i>	<i>st_mtime</i>	Time of last data modification.
645	<i>time_t</i>	<i>st_ctime</i>	Time of last file status change.
646			

647 NOTE: File serial number and device ID taken together uniquely identify the file within the system.

648 All of the described members shall appear in the *stat* structure. The structure
649 members *st_mode*, *st_ino*, *st_dev*, *st_uid*, *st_gid*, *st_atime*, *st_ctime*, and *st_mtime*
650 shall have meaningful values for all file types defined in this part of ISO/IEC 9945.
651 The value of the member *st_nlink* shall be set to the number of links to the file.

652 5.6.1.1 <sys/stat.h> File Types

653 The following macros shall test whether a file is of the specified type. The value
654 *m* supplied to the macros is the value of *st_mode* from a *stat* structure. The macro
655 evaluates to a nonzero value if the test is true, zero if the test is false.

- 656 S_ISDIR(*m*) Test macro for a directory file.
- 657 S_ISCHR(*m*) Test macro for a character special file.
- 658 S_ISBLK(*m*) Test macro for a block special file.
- 659 S_ISREG(*m*) Test macro for a regular file.
- 660 S_ISFIFO(*m*) Test macro for a pipe or a FIFO special file.

661 5.6.1.2 <sys/stat.h> File Modes

662 The file modes portion of values of type *mode_t*, such as the *st_mode* value, are
663 bit-encoded with the following masks and bits:

- 664 S_IRWXU Read, write, search (if a directory), or execute (otherwise) permis-
665 sions mask for the file owner class.
- 666 S_IRUSR Read permission bit for the file owner class.
- 667 S_IWUSR Write permission bit for the file owner class.
- 668 S_IXUSR Search (if a directory) or execute (otherwise) per-
669 missions bit for the file owner class.
- 670 S_IRWXG Read, write, search (if a directory), or execute (otherwise) permis-
671 sions mask for the file group class.
- 672 S_IRGRP Read permission bit for the file group class.
- 673 S_IWGRP Write permission bit for the file group class.
- 674 S_IXGRP Search (if a directory) or execute (otherwise) per-
675 missions bit for the file group class.
- 676 S_IRWXO Read, write, search (if a directory), or execute (otherwise) permis-
677 sions mask for the file other class.
- 678 S_IROTH Read permission bit for the file other class.
- 679 S_IWOTH Write permission bit for the file other class.
- 680 S_IXOTH Search (if a directory) or execute (otherwise) per-
681 missions bit for the file other class.
- 682 S_ISUID Set user ID on execution. The effective user ID of the process
683 shall be set to that of the owner of the file when the file is run as
684 a program (see *exec*). On a regular file, this bit should be cleared
685 on any write.
- 686 S_ISGID Set group ID on execution. Set effective group ID on the process to
687 the group of the file when the file is run as a program (see *exec*).
688 On a regular file, this bit should be cleared on any write.

689 The bits defined by `S_IRUSR`, `S_IWUSR`, `S_IXUSR`, `S_IRGRP`, `S_IWGRP`, `S_IXGRP`,
 690 `S_IROTH`, `S_IWOTH`, `S_IXOTH`, `S_ISUID`, and `S_ISGID` shall be unique. `S_IRWXU`
 691 shall be the bitwise inclusive OR of `S_IRUSR`, `S_IWUSR`, and `S_IXUSR`. `S_IRWXG`
 692 shall be the bitwise inclusive OR of `S_IRGRP`, `S_IWGRP`, and `S_IXGRP`. `S_IRWXO`
 693 shall be the bitwise inclusive OR of `S_IROTH`, `S_IWOTH`, and `S_IXOTH`. Imple-
 694 mentations may OR other implementation-defined bits into `S_IRWXU`, `S_IRWXG`,
 695 and `S_IRWXO`, but they shall not overlap any of the other bits defined in this part
 696 of ISO/IEC 9945. The *file permission bits* are defined to be those corresponding to
 697 the bitwise inclusive OR of `S_IRWXU`, `S_IRWXG`, and `S_IRWXO`.

698 5.6.1.3 <sys/stat.h> Time Entries

699 The time-related fields of *struct stat* are as follows:

700 `st_atime` Accessed file data, for example, *read()*.
 701 `st_mtime` Modified file data, for example, *write()*.
 702 `st_ctime` Changed file status, for example, *chmod()*.

703 These times are updated as described in 2.3.5.

704

705 Times are given in seconds since the Epoch.

706 5.6.1.4 Cross-References

707 *chmod()*, 5.6.4; *chown()*, 5.6.5; *creat()*, 5.3.2; *exec*, 3.1.2; *link()*, 5.3.4; *mkdir()*,
 708 5.4.1; *mkfifo()*, 5.4.2; *pipe()*, 6.1.1; *read()*, 6.4.1; *unlink()*, 5.5.1; *utime()*, 5.6.6;
 709 *write()*, 6.4.2; *remove()* [C Standard {2}].

710 5.6.2 Get File Status

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

712 5.6.2.1 Synopsis

713 `#include <sys/types.h>`
 714 `#include <sys/stat.h>`
 715 `int stat(const char *path, struct stat *buf);`
 716 `int fstat(int fildes, struct stat *buf);`

717 5.6.2.2 Description

718 The *path* argument points to a pathname naming a file. Read, write, or execute
 719 permission for the named file is not required, but all directories listed in the path-
 720 name leading to the file must be searchable. The *stat()* function obtains informa-
 721 tion about the named file and writes it to the area pointed to by the *buf* argument.

722 Similarly, the *fstat()* function obtains information about an open file known by the
 723 file descriptor *fildes*.

724 An implementation that provides additional or alternate file access control
725 mechanisms may, under implementation-defined conditions, cause the *stat()* and
726 *fstat()* functions to fail. In particular, the system may deny the existence of the
727 file specified by *path*.

728 Both functions update any time-related fields, as described in 2.3.5, before writing
729 into the *stat* structure.

730 The *buf* is taken to be a pointer to a *stat* structure, as defined in the header
731 `<sys/stat.h>`, into which information is placed concerning the file.

732 5.6.2.3 Returns

733 Upon successful completion, a value of zero shall be returned. Otherwise, a value
734 of `-1` shall be returned and *errno* shall be set to indicate the error.

735 5.6.2.4 Errors

736 If any of the following conditions occur, the *stat()* function shall return `-1` and set
737 *errno* to the corresponding value:

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

739 [ENAMETOOLONG]

740 The length of the *path* argument exceeds `(PATH_MAX)`, or a
741 pathname component is longer than `(NAME_MAX)` while
742 `{_POSIX_NO_TRUNC}` is in effect.

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

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

746 If any of the following conditions occur, the *fstat()* function shall return `-1` and set
747 *errno* to the corresponding value:

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

749 5.6.2.5 Cross-References

750 *creat()*, 5.3.2; *dup()*, 6.2.1; *fcntl()*, 6.5.2; *open()*, 5.3.1; *pipe()*, 6.1.1;
751 `<sys/stat.h>`, 5.6.1.

752 5.6.3 Check File Accessibility

753 Function: *access()*

754 5.6.3.1 Synopsis

755 `#include <unistd.h>`

756 `int access(const char *path, int amode);`

757 **5.6.3.2 Description**

758 The *access()* function checks the accessibility of the file named by the pathname
 759 pointed to by the *path* argument for the file access permissions indicated by
 760 *amode*, using the real user ID in place of the effective user ID and the real group
 761 ID in place of the effective group ID.

762 The value of *amode* is either the bitwise inclusive OR of the access permissions to
 763 be checked (R_OK, W_OK, and X_OK) or the existence test (F_OK). See 2.9.1 for
 764 the description of these symbolic constants.

765 If any access permission is to be checked, each shall be checked individually, as
 766 described in 2.3.2. If the process has appropriate privileges, an implementation
 767 may indicate success for X_OK even if none of the execute file permission bits are
 768 set.

769 **5.6.3.3 Returns**

770 If the requested access is permitted, a value of zero shall be returned. Otherwise,
 771 a value of -1 shall be returned and *errno* shall be set to indicate the error.

772 **5.6.3.4 Errors**

773 If any of the following conditions occur, the *access()* function shall return -1 and
 774 set *errno* to the corresponding value:

775 [EACCES] The permissions specified by *amode* are denied, or search per-
 776 mission is denied on a component of the path prefix.

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

781 [ENOENT] The *path* argument points to an empty string or to the name of
 782 a file that does not exist.

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

784 [EROFS] Write access was requested for a file residing on a read-only file
 785 system.

786 For each of the following conditions, if the condition is detected, the *access()* func-
 787 tion shall return -1 and set *errno* to the corresponding value:

788 [EINVAL] An invalid value was specified for *amode*.

789 **5.6.3.5 Cross-References**

790 *chmod()*, 5.6.4; *stat()*, 5.6.2; <unistd.h>, 2.9.

91 5.6.4 Change File Modes

92 Function: *chmod()*

93 5.6.4.1 Synopsis

```
94 #include <sys/types.h>
95 #include <sys/stat.h>
96 int chmod(const char *path, mode_t mode);
```

97 5.6.4.2 Description

98 The *path* argument shall point to a pathname naming a file. If the effective user
99 ID of the calling process matches the file owner or the calling process has
100 appropriate privileges, the *chmod()* function shall set the S_ISUID, S_ISGID, and
101 the file permission bits, as described in 5.6.1, of the named file from the
102 corresponding bits in the *mode* argument. These bits define access permissions
103 for the user associated with the file, the group associated with the file, and all oth-
104 ers, as described in 2.3.2. Additional implementation-defined restrictions may
105 cause the S_ISUID and S_ISGID bits in *mode* to be ignored.

106 If the calling process does not have appropriate privileges, if the group ID of the
107 file does not match the effective group ID or one of the supplementary group IDs,
108 and if the file is a regular file, bit S_ISGID (set group ID on execution) in the mode
109 of the file shall be cleared upon successful return from *chmod()*.

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

112 Upon successful completion, the *chmod()* function shall mark for update the
113 *st_ctime* field of the file.

114 5.6.4.3 Returns

115 Upon successful completion, the function shall return a value of zero. Otherwise,
116 a value of -1 shall be returned and *errno* shall be set to indicate the error. If -1 is
117 returned, no change to the file mode shall have occurred.

118 5.6.4.4 Errors

119 If any of the following conditions occur, the *chmod()* function shall return -1 and
120 set *errno* to the corresponding value:

- 1 [EACCES] Search permission is denied on a component of the path prefix.
- 2 [ENAMETOOLONG] The length of the *path* argument exceeds {PATH_MAX}, or a
3 pathname component is longer than {NAME_MAX} while
4 [_POSIX_NO_TRUNC] is in effect.
- 5 [ENOTDIR] A component of the path prefix is not a directory.

- 827 [ENOENT] The named file does not exist or the *path* argument points to an
828 empty string.
- 829 [EPERM] The effective user ID does not match the owner of the file, and
830 the calling process does not have the appropriate privileges.
- 831 [EROFS] The named file resides on a read-only file system.

832 5.6.4.5 Cross-References

833 *chown()*, 5.6.5; *mkdir()*, 5.4.1; *mkfifo()*, 5.4.2; *stat()*, 5.6.2; <sys/stat.h>, 5.6.1.

834 5.6.5 Change Owner and Group of a File

835 Function: *chown()*

836 5.6.5.1 Synopsis

```
837   #include <sys/types.h>
838   int chown(const char *path, uid_t owner, gid_t group);
```

839 5.6.5.2 Description

840 The *path* argument points to a pathname naming a file. The user ID and group ID
841 of the named file are set to the numeric values contained in *owner* and *group*
842 respectively.

843 Only processes with an effective user ID equal to the user ID of the file or
844 with appropriate privileges may change the ownership of a file. If
845 (_POSIX_CHOWN_RESTRICTED) is in effect for *path*:

- 846 (1) Changing the owner is restricted to processes with appropriate
847 privileges.
- 848 (2) Changing the group is permitted to a process without appropriate
849 privileges, but with an effective user ID equal to the user ID of the file, if
850 and only if *owner* is equal to the user ID of the file and *group* is equal
851 either to the effective group ID of the calling process or to one of its sup-
852 plementary group IDs.

853 If the *path* argument refers to a regular file, the set-user-ID (S_ISUID) and set-
854 group-ID (S_ISGID) bits of the file mode shall be cleared upon successful return
855 from *chown()*, unless the call is made by a process with appropriate privileges, in
856 which case it is implementation defined whether those bits are altered. If the
857 *chown()* function is successfully invoked on a file that is not a regular file, these
858 bits may be cleared. These bits are defined in 5.6.1.

859 Upon successful completion, the *chown()* function shall mark for update the
860 *st_ctime* field of the file.

5.6.5.3 Returns

Upon successful completion, a value of zero shall be returned. Otherwise, a value of `-1` shall be returned and *errno* shall be set to indicate the error. If `-1` is returned, no change shall be made in the owner and group of the file.

5.6.5.4 Errors

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

- [EACCES] Search permission is denied on a component of the path prefix.
- [ENAMETOOLONG] The length of the *path* argument exceeds `{PATH_MAX}`, or a pathname component is longer than `{NAME_MAX}` while `{_POSIX_NO_TRUNC}` is in effect.
- [ENOTDIR] A component of the path prefix is not a directory.
- [ENOENT] The named file does not exist, or the *path* argument points to an empty string.
- [EPERM] The effective user ID does not match the owner of the file, or the calling process does not have appropriate privileges and `{_POSIX_CHOWN_RESTRICTED}` indicates that such privilege is required.
- [EROFS] The named file resides on a read-only file system.

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

- [EINVAL] The owner or group ID supplied is invalid and not supported by the implementation.

5.6.5.5 Cross-References

chmod(), 5.6.4; `<sys/stat.h>`, 5.6.1.

5.6.6 Set File Access and Modification Times

Function: *utime()*

5.6.6.1 Synopsis

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

int utime(const char *path, const struct utimbuf *times);
```

893 **5.6.6.2 Description**

894 The argument *path* points to a pathname naming a file. The *utime()* function sets
895 the access and modification times of the named file.

896 If the *times* argument is *NULL*, the access and modification times of the file are
897 set to the current time. The effective user ID of the process must match the owner
898 of the file, or the process must have write permission to the file or appropriate
899 privileges, to use the *utime()* function in this manner.

900 If the *times* argument is not *NULL*, it is interpreted as a pointer to a *utimbuf*
901 structure, and the access and modification times are set to the values contained in
902 the designated structure. Only the owner of the file and processes with appropri-
903 ate privileges shall be permitted to use the *utime()* function in this way.

904 The *utimbuf* structure is defined by the header `<utime.h>` and includes the fol-
905 lowing members:

906	Member	Member	Description
907	Type	Name	
908	<i>time_t</i>	<i>actime</i>	Access time
909	<i>time_t</i>	<i>modtime</i>	Modification time

910 The times in the *utimbuf* structure are measured in seconds since the Epoch.

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

915 Upon successful completion, the *utime()* function shall mark for update the
916 *st_ctime* field of the file.

917 **5.6.6.3 Returns**

918 Upon successful completion, the function shall return a value of zero. Otherwise,
919 a value of -1 shall be returned, *errno* is set to indicate the error, and the file times
920 shall not be affected.

921 **5.6.6.4 Errors**

922 If any of the following conditions occur, the *utime()* function shall return -1 and
923 set *errno* to the corresponding value:

924 [EACCES] Search permission is denied by a component of the path prefix,
925 or the *times* argument is *NULL* and the effective user ID of the
926 process does not match the owner of the file and write access is
927 denied.

928 [ENAMETOOLONG]

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

- 2 [ENOENT] The named file does not exist or the *path* argument points to an
- 3 empty string.
- 4 [ENOTDIR] A component of the path prefix is not a directory.
- 5 [EPERM] The *times* argument is not **NULL**, the effective user ID of the
- 6 calling process has write access to the file, but does not match
- 7 the owner of the file, and the calling process does not have the
- 8 appropriate privileges.
- 9 [EROFS] The *named* file resides on a read-only file system.

0 **5.6.6.5 Cross-References**

1 <sys/stat.h>, 5.6.1.

2 **5.7 Configurable Pathname Variables**

3 **5.7.1 Get Configurable Pathname Variables**

4 Functions: *pathconf()*, *fpathconf()*

5 **5.7.1.1 Synopsis**

```
6  #include <unistd.h>
7  long pathconf(const char *path, int name);
8  long fpathconf(int fildes, int name);
```

9 **5.7.1.2 Description**

10 The *pathconf()* and *fpathconf()* functions provide a method for the application to
11 determine the current value of a configurable limit or option (*variable*) that is
12 associated with a file or directory.

13 For *pathconf()*, the *path* argument points to the pathname of a file or directory.
14 For *fpathconf()*, the *fildes* argument is an open file descriptor.

15 The *name* argument represents the variable to be queried relative to that file or
16 directory. The implementation shall support all of the variables listed in
17 Table 5-2 and may support others. The variables in Table 5-2 come from
18 <limits.h> or <unistd.h> and the symbolic constants, defined in
19 <unistd.h>, that are the corresponding values used for *name*.

10 **5.7.1.3 Returns**

11 If *name* is an invalid value, the *pathconf()* and *fpathconf()* functions shall
12 return **-1**.

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

Table 5-2 – Configurable Pathname Variables

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

NOTES:

- (1) If *path* or *fildev* refers to a directory, the value returned applies to the directory itself.
- (2) If *path* or *fildev* does not refer to a terminal file, it is unspecified whether an implementation supports an association of the variable name with the specified file.
- (3) If *path* or *fildev* refers to a directory, the value returned applies to the filenames within the directory.
- (4) If *path* or *fildev* does not refer to a directory, it is unspecified whether an implementation supports an association of the variable name with the specified file.
- (5) If *path* or *fildev* 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 *fildev* refers to a pipe or a FIFO, the value returned applies to the referenced object itself. If *path* or *fildev* refers to a directory, the value returned applies to any FIFOs that exist or can be created within the directory. If *path* or *fildev* refers to any other type of file, it is unspecified whether an implementation supports an association of the variable name with the specified file.
- (7) If *path* or *fildev* refers to a directory, the value returned applies to any files defined in this part of ISO/IEC 9945, other than directories, that exist or can be created within the directory.

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

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

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

08 **5.7.1.4 Errors**

09 If any of the following conditions occur, the *pathconf()* and *fpathconf()* functions
10 shall return `-1` and set *errno* to the corresponding value:

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

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

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

15 [EINVAL] The implementation does not support an association of the vari-
16 able name with the specified file.

17 [ENAMETOOLONG]

18 The length of the *path* argument exceeds `{PATH_MAX}`, or a
19 pathname component is longer than `{NAME_MAX}` while
20 `{_POSIX_NO_TRUNC}` is in effect.

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

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

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

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

27 [EINVAL] The implementation does not support an association of the vari-
28 able name with the specified file.

Section 6: Input and Output Primitives

1 The functions in this section deal with input and output from files and pipes. |
2 Functions are also specified that deal with the coordination and management of
3 file descriptors and I/O activity.

4 6.1 Pipes

5 6.1.1 Create an Inter-Process Channel

6 Function: *pipe()*

7 6.1.1.1 Synopsis

8 `int pipe(int fildes[2]);` |

9 6.1.1.2 Description

10 The *pipe()* function shall create a pipe and place two file descriptors, one each into
11 the arguments *fildes*[0] and *fildes*[1], that refer to the open file descriptions for
12 the read and write ends of the pipe. Their integer values shall be the two lowest
13 available at the time of the *pipe()* function call. The O_NONBLOCK and |
14 FD_CLOEXEC flags shall be clear on both file descriptors. [The *fcntl()* function |
15 can be used to set these flags.]

16 Data can be written to file descriptor *fildes*[1] and read from file descriptor
17 *fildes*[0]. A read on file descriptor *fildes*[0] shall access the data written to file
18 descriptor *fildes*[1] on a first-in-first-out basis.

19 A process has the pipe open for reading if it has a file descriptor open that refers
20 to the read end, *fildes*[0]. A process has the pipe open for writing if it has a file
21 descriptor open that refers to the write end, *fildes*[1].

22 Upon successful completion, the *pipe()* function shall mark for update the
23 *st_atime*, *st_ctime*, and *st_mtime* fields of the pipe.

24 6.1.1.3 Returns

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

6.1.1.4 Errors

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

- [EMFILE] More than {OPEN_MAX}-2 file descriptors are already in use by this process.
- [ENFILE] The number of simultaneously open files in the system would exceed a system-imposed limit.

6.1.1.5 Cross-References

fcntl(), 6.5.2; *open()*, 5.3.1; *read()*, 6.4.1; *write()*, 6.4.2.

6.2 File Descriptor Manipulation

6.2.1 Duplicate an Open File Descriptor

Functions: *dup()*, *dup2()*

6.2.1.1 Synopsis

```
int dup(int fildes);
int dup2(int fildes, int fildes2);
```

6.2.1.2 Description

The *dup()* and *dup2()* functions provide an alternate interface to the service provided by the *fcntl()* function using the F_DUPFD command. The call:

```
fid = dup (fildes);
```

shall be equivalent to:

```
fid = fcntl (fildes, F_DUPFD, 0);
```

The call:

```
fid = dup2 (fildes, fildes2);
```

shall be equivalent to:

```
close (fildes2);
fid = fcntl (fildes, F_DUPFD, fildes2);
```

except for the following:

- (1) If *fildes2* is negative or greater than or equal to {OPEN_MAX}, the *dup2()* function shall return *-1* and *errno* shall be set to [EBADF].
- (2) If *fildes* is a valid file descriptor and is equal to *fildes2*, the *dup2()* function shall return *fildes2* without closing it.

- 58 (3) If *fildest* is not a valid file descriptor, *dup2()* shall fail and not close
59 *fildest2*.
- 60 (4) The value returned shall be equal to the value of *fildest2* upon successful
61 completion or shall be -1 upon failure. |

62 6.2.1.3 Returns

63 Upon successful completion, the function shall return a file descriptor. Other-
64 wise, a value of -1 shall be returned and *errno* shall be set to indicate the error.

65 6.2.1.4 Errors

66 If any of the following conditions occur, the *dup()* function shall return -1 and set
67 *errno* to the corresponding value: |

68 [EBADF] The argument *fildest* is not a valid open file descriptor. |

69 [EMFILE] The number of file descriptors would exceed (OPEN_MAX). |

70 If any of the following conditions occur, the *dup2()* function shall return -1 and
71 set *errno* to the corresponding value: |

72 [EBADF] The argument *fildest* is not a valid open file descriptor, or the
73 argument *fildest2* is negative or greater than or equal to
74 (OPEN_MAX). |

75 [EINTR] The *dup2()* function was interrupted by a signal. |

76 6.2.1.5 Cross-References

77 *close()*, 6.3.1; *creat()*, 5.3.2; *exec*, 3.1.2; *fcntl()*, 6.5.2; *open()*, 5.3.1; *pipe()*, 6.1.1.

78 6.3 File Descriptor Deassignment

79 6.3.1 Close a File

80 Function: *close()*

81 6.3.1.1 Synopsis

82 int *close*(int *fildest*); |

83 6.3.1.2 Description

84 The *close()* function shall deallocate (i.e., make available for return by subsequent
85 *open()*s, etc., executed by the process) the file descriptor indicated by *fildest*. All
86 outstanding record locks owned by the process on the file associated with the file
87 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 *fil* 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 *fil* argument is not a valid file descriptor.
- [EINTR] The *close* function was interrupted by a signal.

6.3.1.5 Cross-References

creat(), 5.3.2; *dup()*, 6.2.1; *exec*, 3.1.2; *fcntl()*, 6.5.2; *fork()*, 3.1.1; *open()*, 5.3.1; *pipe()*, 6.1.1; *unlink()*, 5.5.1; 3.3.1.4.

6.4 Input and Output

6.4.1 Read from a File

Function: *read()*

6.4.1.1 Synopsis

```
ssize_t read(int fil, void *buf, size_t nbyte);
```

6.4.1.2 Description

The *read()* function shall attempt to read *nbyte* bytes from the file associated with the open file descriptor, *fil*, 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 *fil*. Before successful return

119 from *read()*, the file offset shall be incremented by the number of bytes actually
120 read.

121 On a file not capable of seeking, the *read()* shall start from the current position.
122 The value of a file offset associated with such a file is undefined.

123 Upon successful completion, the *read()* function shall return the number of bytes
124 actually read and placed in the buffer. This number shall never be greater than
125 *nbyte*. The value returned may be less than *nbyte* if the number of bytes left in
126 the file is less than *nbyte*, if the *read()* request was interrupted by a signal, or if
127 the file is a pipe (or FIFO) or special file and has fewer than *nbyte* bytes immedi-
128 ately available for reading. For example, a *read()* from a file associated with a
129 terminal may return one typed line of data.

130 If a *read()* is interrupted by a signal before it reads any data, it shall return -1
131 with *errno* set to [EINTR].

132 If a *read()* is interrupted by a signal after it has successfully read some data,
133 either it shall return -1 with *errno* set to [EINTR], or it shall return the number of
134 bytes read. A *read()* from a pipe or FIFO shall never return with *errno* set to
135 [EINTR] if it has transferred any data.

136 No data transfer shall occur past the current end-of-file. If the starting position is
137 at or after the end-of-file, zero shall be returned. If the file refers to a device spe-
138 cial file, the result of subsequent *read()* requests is implementation defined.

139 If the value of *nbyte* is greater than {SSIZE_MAX}, the result is implementation
140 defined. |

141 When attempting to read from an empty pipe (or FIFO):

142 (1) If no process has the pipe open for writing, *read()* shall return zero to
143 indicate end-of-file.

144 (2) If some process has the pipe open for writing and O_NONBLOCK is set,
145 *read()* shall return -1 and set *errno* to [EAGAIN].

146 (3) If some process has the pipe open for writing and O_NONBLOCK is clear, |
147 *read()* shall block until some data is written or the pipe is closed by all |
148 processes that had the pipe open for writing. |

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

151 (1) If O_NONBLOCK is set, *read()* shall return -1 and set *errno* to [EAGAIN].

152 (2) If O_NONBLOCK is clear, *read()* shall block until some data becomes
153 available.

154 The use of the O_NONBLOCK flag has no effect if there is some data available.

155 For any portion of a regular file, prior to the end-of-file, that has not been written,
156 *read()* shall return bytes with value zero.

157 Upon successful completion where *nbyte* is greater than zero, the *read()* function |
158 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 *fil-des* argument is not a valid file descriptor open for reading.
- [EINTR] The read operation was interrupted by a signal, and either no data was transferred or the implementation does not report partial transfer for this file.
- [EIO] The implementation supports job control, the process is in a background process group and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the SIGTTIN signal or the process group of the process is orphaned. This error may also be generated when conditions unspecified by this part of ISO/IEC 9945 occur.

6.4.1.5 Cross-References

creat(), 5.3.2; *dup()*, 6.2.1; *fcntl()*, 6.5.2; *lseek()*, 6.5.3; *open()*, 5.3.1; *pipe()*, 6.1.1; 3.3.1.4; 7.1.1.

6.4.2 Write to a File

Function: *write()*

6.4.2.1 Synopsis

```
ssize_t write(int fil-des, const void *buf, size_t nbyte);
```

6.4.2.2 Description

The *write()* function shall attempt to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the open file descriptor, *fil-des*.

If *nbyte* is zero and the file is a regular file, the *write()* function shall return zero and have no other results. If *nbyte* is zero and the file is not a regular file, the results are unspecified.

On a regular file or other file capable of seeking, the actual writing of data shall proceed from the position in the file indicated by the file offset associated with

195 *files*. Before successful return from *write()*, the file offset shall be incremented
 196 by the number of bytes actually written. On a regular file, if this incremented file
 197 offset is greater than the length of the file, the length of the file shall be set to this
 198 file offset.

199 On a file not capable of seeking, the *write()* shall start from the current position.
 200 The value of a file offset associated with such a file is undefined.

201 If the *O_APPEND* flag of the file status flags is set, the file offset shall be set to the
 202 end of the file prior to each write, and no intervening file modification operation
 203 shall be allowed between changing the file offset and the write operation.

204 If a *write()* requests that more bytes be written than there is room for (for exam-
 205 ple, the physical end of a medium), only as many bytes as there is room for shall
 206 be written. For example, suppose there is space for 20 bytes more in a file before
 207 reaching a limit. A write of 512 bytes would return 20. The next write of a
 208 nonzero number of bytes would give a failure return (except as noted below).

209 Upon successful completion, the *write()* function shall return the number of bytes
 210 actually written to the file associated with *files*. This number shall never be
 211 greater than *nbyte*.

212 If a *write()* is interrupted by a signal before it writes any data, it shall return -1
 213 with *errno* set to [EINTR].

214 If *write()* is interrupted by a signal after it successfully writes some data, either it
 215 shall return -1 with *errno* set to [EINTR], or it shall return the number of bytes
 216 written. A *write()* to a pipe or FIFO shall never return with *errno* set to [EINTR] if
 217 it has transferred any data and *nbyte* is less than or equal to {PIPE_BUF}.

218 If the value of *nbyte* is greater than {SSIZE_MAX}, the result is implementation
 219 defined.

220 After a *write()* to a regular file has successfully returned:

- 221 (1) Any successful *read()* from each byte position in the file that was
 222 modified by that *write()* shall return the data specified by the *write()* for
 223 that position, until such byte positions are again modified.
- 224 (2) Any subsequent successful *write()* to the same byte position in the file
 225 shall overwrite that file data. The phrase "subsequent successful *write()*"
 226 in the previous sentence is intended to be viewed from a system perspec-
 227 tive [i.e., *read()* followed by a systemwide subsequent *write()*].

228 Write requests to a pipe (or FIFO) shall be handled in the same manner as write
 229 requests to a regular file, with the following exceptions:

- 230 (1) There is no file offset associated with a pipe, hence each write request
 231 shall append to the end of the pipe.
- 232 (2) Write requests of {PIPE_BUF} bytes or less shall not be interleaved with
 233 data from other processes doing writes on the same pipe. Writes of
 234 greater than {PIPE_BUF} bytes may have data interleaved, on arbitrary
 235 boundaries, with writes by other processes, whether or not the
 236 *O_NONBLOCK* flag of the file status flags is set.

- (3) If the `O_NONBLOCK` flag is clear, a write request may cause the process to block, but on normal completion it shall return *nbyte*.
- (4) If the `O_NONBLOCK` flag is set, *write()* requests shall be handled differently, in the following ways:
 - (a) The *write()* function shall not block the process.
 - (b) A write request for `{PIPE_BUF}` or fewer bytes shall either:
 - [1] If there is sufficient space available in the pipe, transfer all the data and return the number of bytes requested.
 - [2] If there is not sufficient space available in the pipe, transfer no data and return `-1` with *errno* set to `[EAGAIN]`.
 - (c) A write request for more than `{PIPE_BUF}` bytes shall either:
 - [1] When at least one byte can be written, transfer what it can and return the number of bytes written. When all data previously written to the pipe has been read, it shall transfer at least `{PIPE_BUF}` bytes.
 - [2] When no data can be written, transfer no data and return `-1` with *errno* set to `[EAGAIN]`.

When attempting to write to a file descriptor (other than a pipe or FIFO) that supports nonblocking writes and cannot accept the data immediately:

- (1) If the `O_NONBLOCK` flag is clear, *write()* shall block until the data can be accepted.
- (2) If the `O_NONBLOCK` flag is set, *write()* shall not block the process. If some data can be written without blocking the process, *write()* shall write what it can and return the number of bytes written. Otherwise, it shall return `-1` and *errno* shall be set to `[EAGAIN]`.

Upon successful completion where *nbyte* is greater than zero, the *write()* function shall mark for update the *st_ctime* and *st_mtime* fields of the file.

6.4.2.3 Returns

Upon successful completion, *write()* shall return an integer indicating the number of bytes actually written. Otherwise, it shall return a value of `-1` and set *errno* to indicate the error.

6.4.2.4 Errors

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

- | | |
|-----------------------|--|
| <code>[EAGAIN]</code> | The <code>O_NONBLOCK</code> flag is set for the file descriptor and the process would be delayed in the write operation. |
| <code>[EBADF]</code> | The <i>fdes</i> argument is not a valid file descriptor open for writing. |

275 [EFBIG] An attempt was made to write a file that exceeds an
276 implementation-defined maximum file size.

277 [EINTR] The write operation was interrupted by a signal, and either no
278 data was transferred or the implementation does not report
279 partial transfers for this file.

280 [EIO] The implementation supports job control, the process is in a
281 background process group and is attempting to write to its con-
282 trolling terminal, TOSTOP is set, the process is neither ignoring
283 nor blocking SIGTTOU signals, and the process group of the pro-
284 cess is orphaned. This error may also be generated when condi-
285 tions unspecified by this part of ISO/IEC 9945 occur.

286 [ENOSPC] There is no free space remaining on the device containing the
287 file.

288 [EPIPE] An attempt is made to write to a pipe (or FIFO) that is not open
289 for reading by any process. A SIGPIPE signal shall also be sent
290 to the process.

291 **6.4.2.5 Cross-References**

292 *creat()*, 5.3.2; *dup()*, 6.2.1; *fcntl()*, 6.5.2; *lseek()*, 6.5.3; *open()*, 5.3.1; *pipe()*, 6.1.1;
293 3.3.1.4.

294 **6.5 Control Operations on Files**

295 **6.5.1 Data Definitions for File Control Operations**

296 The header `<fcntl.h>` defines the following *requests* and *arguments* for the
297 *fcntl()* and *open()* functions. The values within each of the tables within this
298 clause (Table 6-1 through Table 6-7) shall be unique numbers. In addition, the
299 values of the entries for *oflag* values, file status flags, and file access modes shall
300 be unique.

301 **6.5.2 File Control**

302 Function: *fcntl()*

303 **6.5.2.1 Synopsis**

```
304    #include <sys/types.h>
305    #include <unistd.h>
306    #include <fcntl.h>
307    int fcntl(int fd, int cmd, ...);
```

Table 6-1 – *cmd* Values for *fcntl()*

Constant	Description
F_DUPFD	Duplicate file descriptor.
F_GETFD	Get file descriptor flags.
F_GETLK	Get record locking information.
F_SETFD	Set file descriptor flags.
F_GETFL	Get file status flags.
F_SETFL	Set file status flags.
F_SETLK	Set record locking information.
F_SETLKW	Set record locking information; wait if blocked.

Table 6-2 – File Descriptor Flags Used for *fcntl()*

Constant	Description
FD_CLOEXEC	Close the file descriptor upon execution of an <i>exec</i> -family function.

Table 6-3 – *l_type* Values for Record Locking With *fcntl()*

Constant	Description
F_RDLCK	Shared or read lock.
F_UNLCK	Unlock.
F_WRLCK	Exclusive or write lock.

Table 6-4 – *oflag* Values for *open()*

Constant	Description
O_CREAT	Create file if it does not exist.
O_EXCL	Exclusive use flag.
O_NOCTTY	Do not assign a controlling terminal.
O_TRUNC	Truncate flag.

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

Constant	Description
O_APPEND	Set append mode.
O_NONBLOCK	No delay.

Table 6-6 – File Access Modes Used for *open()* and *fcntl()*

Constant	Description
O_RDONLY	Open for reading only.
O_RDWR	Open for reading and writing.
O_WRONLY	Open for writing only.

Table 6-7 – Mask for Use With File Access Modes

Constant	Description
O_ACCMODE	Mask for file access modes.

6.5.2.2 Description

The function *fcntl()* provides for control over open files. The argument *fil-des* is a file descriptor.

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

F_DUPFD	Return a new file descriptor that is the lowest numbered available (i.e., not already open) file descriptor greater than or equal to the third argument, <i>arg</i> , taken as an integer of type <i>int</i> . 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 <i>exec</i> family of functions.
F_GETFD	Get the file descriptor flags, as defined in Table 6-2, that are associated with the file descriptor <i>fil-des</i> . File descriptor flags are associated with a single file descriptor and do not affect other file descriptors that refer to the same file.
F_SETFD	Set the file descriptor flags, as defined in Table 6-2, that are associated with <i>fil-des</i> to the third argument, <i>arg</i> , taken as type <i>int</i> . If the FD_CLOEXEC flag is zero, the file shall remain open across <i>exec</i> functions; otherwise, the file shall be closed upon successful execution of an <i>exec</i> function.
F_GETFL	Get the file status flags, as defined in Table 6-5, and file access modes for the open file description associated with <i>fil-des</i> . 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 <code><fcntl.h></code> . File status flags and file access modes are associated with the open file description and do not affect other file descriptors that refer to the same file with different open file descriptions.

F_SETFL Set the file status flags, as defined in Table 6-5, for the open file description associated with *fdes* from the corresponding bits in the third argument, *arg*, taken as type *int*. Bits corresponding to the file access modes (as defined in Table 6-6) and the *oflag* values (as defined in Table 6-4) that are set in *arg* are ignored. If any bits in *arg* other than those mentioned here are changed by the application, the result is unspecified.

The following commands are available for advisory record locking. Advisory record locking shall be supported for regular files, and may be supported for other files.

F_GETLK Get the first lock that blocks the lock description pointed to by the third argument, *arg*, taken as a pointer to type *struct flock* (see below). The information retrieved overwrites the information passed to *fcntl()* in the *flock* structure. If no lock is found that would prevent this lock from being created, the structure shall be left unchanged by this function call except for the lock type, which shall be set to **F_UNLCK**.

F_SETLK Set or clear a file segment lock according to the lock description pointed to by the third argument, *arg*, taken as a pointer to type *struct flock* (see below). **F_SETLK** is used to establish shared (or read) locks (**F_RDLCK**) or exclusive (or write) locks, (**F_WRLCK**), as well as to remove either type of lock (**F_UNLCK**). **F_RDLCK**, **F_WRLCK**, and **F_UNLCK** are defined by the *<fcntl.h>* header. If a shared or exclusive lock cannot be set, *fcntl()* shall return immediately.

F_SETLKW This command is the same as **F_SETLK** except that if a shared or exclusive lock is blocked by other locks, the process shall wait until the request can be satisfied. If a signal that is to be caught is received while *fcntl()* is waiting for a region, the *fcntl()* shall be interrupted. Upon return from the signal handler of the process, *fcntl()* shall return *-1* with *errno* set to **[EINTR]**, and the lock operation shall not be done.

The *flock* structure, defined by the *<fcntl.h>* header, describes an advisory lock. It includes the members shown in Table 6-8.

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

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

The value of *l_whence* is **SEEK_SET**, **SEEK_CUR**, or **SEEK_END** to indicate that the relative offset, *l_start* bytes, will be measured from the start of the file, current position, or end of the file, respectively. The value of *l_len* is the number of consecutive bytes to be locked. If *l_len* is negative, the result is undefined. The

Table 6-8 – *flock* Structure

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

l_pid field is only used with F_GETLK to return the process ID of the process holding a blocking lock. After a successful F_GETLK request, the value of *l_whence* shall be SEEK_SET.

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

There shall be at most one type of lock set for each byte in the file. Before a successful return from an F_SETLK or an F_SETLKW request when the calling process has previously existing locks on bytes in the region specified by the request, the previous lock type for each byte in the specified region shall be replaced by the new lock type. As specified above under the descriptions of shared locks and exclusive locks, an F_SETLK or an F_SETLKW request shall (respectively) fail or block when another process has existing locks on bytes in the specified region and the type of any of those locks conflicts with the type specified in the request.

All locks associated with a file for a given process shall be removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process created using the *fork()* function.

A potential for deadlock occurs if a process controlling a locked region is put to sleep by attempting to lock the locked region of another process. If the system detects that sleeping until a locked region is unlocked would cause a deadlock, the *fcntl()* function shall fail with an [EDEADLK] error.

6.5.2.3 Returns

Upon successful completion, the value returned shall depend on *cmd*. The various return values are shown in Table 6-9.

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

Table 6-9 – *fcntl()* Return Values

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

6.5.2.4 Errors

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

[EACCES] or [EAGAIN]

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

[EBADF]

The *fildev* 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 *fildev* 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 *fildev* is not a valid file descriptor open for writing.

[EINTR]

The argument *cmd* is F_SETLKW, and the function was interrupted by a signal.

[EINVAL]

The argument *cmd* is F_DUPFD, and the third argument is negative or greater than or equal to {OPEN_MAX}.

The argument *cmd* is F_GETLK, F_SETLK, or F_SETLKW and the data to which *arg* points is not valid, or *fildev* 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.

516 For each of the following conditions, if the condition is detected, the *fcntl()* func-
517 tion shall return -1 and set *errno* to the corresponding value:

518 [EDEADLK] The argument *cmd* is *F_SETLK*, and a deadlock condition was
519 detected.

520 6.5.2.5 Cross-References

521 *close()*, 6.3.1; *exec*, 3.1.2; *open()*, 5.3.1; *<fcntl.h>*, 6.5.1; 3.3.1.4.

522 6.5.3 Reposition Read/Write File Offset

523 Function: *lseek()*

524 6.5.3.1 Synopsis

```
525 #include <sys/types.h>
526 #include <unistd.h>
527 off_t lseek(int fildev, off_t offset, int whence);
```

528 6.5.3.2 Description

529 The *fildev* argument is an open file descriptor. The *lseek()* function shall set the
530 file offset for the open file description associated with *fildev* as follows:

- 531 (1) If *whence* is *SEEK_SET*, the offset is set to *offset* bytes.
- 532 (2) If *whence* is *SEEK_CUR*, the offset is set to its current value plus *offset*
533 bytes.
- 534 (3) If *whence* is *SEEK_END*, the offset is set to the size of the file plus *offset*
535 bytes.

536 The symbolic constants *SEEK_SET*, *SEEK_CUR*, and *SEEK_END* are defined in the
537 header *<unistd.h>*.

538 Some devices are incapable of seeking. The value of the file offset associated with
539 such a device is undefined. The behavior of the *lseek()* function on such devices is
540 implementation defined.

541 The *lseek()* function shall allow the file offset to be set beyond the end of existing
542 data in the file. If data is later written at this point, subsequent reads of data in
543 the gap shall return bytes with the value zero until data is actually written into
544 the gap.

545 The *lseek()* function shall not, by itself, extend the size of a file.

546 6.5.3.3 Returns

547 Upon successful completion, the function shall return the resulting offset location
548 as measured in bytes from the beginning of the file. Otherwise, it shall return a
549 value of $((\text{off_t}) - 1)$, shall set *errno* to indicate the error, and the file offset shall
550 remain unchanged by this function call.

1 **6.5.3.4 Errors**

2 If any of the following conditions occur, the *lseek()* function shall return *-1* and
3 set *errno* to the corresponding value:

- 4 [EBADF] The *fildev* argument is not a valid file descriptor.
- 5 [EINVAL] The *whence* argument is not a proper value, or the resulting file
6 offset would be invalid.
- 7 [ESPIPE] The *fildev* argument is associated with a pipe or FIFO.

8 **6.5.3.5 Cross-References**

9 *creat()*, 5.3.2; *dup()*, 6.2.1; *fcntl()*, 6.5.2; *open()*, 5.3.1; *read()*, 6.4.1; *sigaction()*,
0 3.3.4; *write()*, 6.4.2; <unistd.h>, 2.9.

Section 7: Device- and Class-Specific Functions

1 **7.1 General Terminal Interface**

2 This section describes a general terminal interface that shall be provided. It shall
3 be supported on any asynchronous communication ports if the implementation
4 provides them. It is implementation defined whether this interface supports net-
5 work connections or synchronous ports or both. The conformance document shall
6 describe which device types are supported by these interfaces. Certain functions
7 in this section apply only to the controlling terminal of a process; where this is the
8 case, it is so noted.

9 **7.1.1 Interface Characteristics**

10 **7.1.1.1 Opening a Terminal Device File**

11 When a terminal file is opened, it normally causes the process to wait until a con-
12 nection is established. In practice, application programs seldom open these files;
13 they are opened by special programs and become the standard input, output, and
14 error files of an application.

15 As described in 5.3.1, opening a terminal device file with the `O_NONBLOCK` flag
16 clear shall cause the process to block until the terminal device is ready and avail-
17 able. The `CLOCAL` flag can also affect *open()*. See 7.1.2.4.

18 **7.1.1.2 Process Groups**

19 A terminal may have a foreground process group associated with it. This fore-
20 ground process group plays a special role in handling signal-generating input
21 characters, as discussed below in 7.1.1.9.

22 If the implementation supports job control (if `{_POSIX_JOB_CONTROL}` is defined;
23 see 2.9), command interpreter processes supporting job control can allocate the
24 terminal to different *jobs*, or process groups, by placing related processes in a sin-
25 gle process group and associating this process group with the terminal. The fore-
26 ground process group of a terminal may be set or examined by a process, assum-
27 ing the permission requirements in this section are met; see 7.2.3 and 7.2.4. The
28 terminal interface aids in this allocation by restricting access to the terminal by
29 processes that are not in the foreground process group; see 7.1.1.4.

30 When there is no longer any process whose process ID or process group ID
31 matches the process group ID of the foreground process group, the terminal shall
32 have no foreground process group. It is unspecified whether the terminal has a
33 foreground process group when there is no longer any process whose process

group ID matches the process group ID of the foreground process group, but there is a process whose process ID matches. No actions defined by this part of ISO/IEC 9945, other than allocation of a controlling terminal as described in 7.1.1.3 or a successful call to *tcsetpgrp()*, shall cause a process group to become the foreground process group of a terminal.

7.1.1.3 The Controlling Terminal

A terminal may belong to a process as its controlling terminal. Each process of a session that has a controlling terminal has the same controlling terminal. A terminal may be the controlling terminal for at most one session. The controlling terminal for a session is allocated by the session leader in an implementation-defined manner. If a session leader has no controlling terminal and opens a terminal device file that is not already associated with a session without using the *O_NOCTTY* option (see 5.3.1), it is implementation defined whether the terminal becomes the controlling terminal of the session leader. If a process that is not a session leader opens a terminal file, or the *O_NOCTTY* option is used on *open()*, that terminal shall not become the controlling terminal of the calling process. When a controlling terminal becomes associated with a session, its foreground process group shall be set to the process group of the session leader.

The controlling terminal is inherited by a child process during a *fork()* function call. A process relinquishes its controlling terminal when it creates a new session with the *setsid()* function; other processes remaining in the old session that had this terminal as their controlling terminal continue to have it. Upon the close of the last file descriptor in the system (whether or not it is in the current session) associated with the controlling terminal, it is unspecified whether all processes that had that terminal as their controlling terminal cease to have any controlling terminal. Whether and how a session leader can reacquire a controlling terminal after the controlling terminal has been relinquished in this fashion is unspecified. A process does not relinquish its controlling terminal simply by closing all of its file descriptors associated with the controlling terminal if other processes continue to have it open.

When a controlling process terminates, the controlling terminal is disassociated from the current session, allowing it to be acquired by a new session leader. Subsequent access to the terminal by other processes in the earlier session may be denied, with attempts to access the terminal treated as if modem disconnect had been sensed.

7.1.1.4 Terminal Access Control

If a process is in the foreground process group of its controlling terminal, read operations shall be allowed as described in 7.1.1.5. For those implementations that support job control, any attempts by a process in a background process group to read from its controlling terminal shall cause its process group to be sent a *SIGTTIN* signal unless one of the following special cases apply: If the reading process is ignoring or blocking the *SIGTTIN* signal, or if the process group of the reading process is orphaned, the *read()* returns *-1* with *errno* set to *[EIO]*, and no signal is sent. The default action of the *SIGTTIN* signal is to stop the process to which it is sent. See 3.3.1.1.

79 If a process is in the foreground process group of its controlling terminal, write
80 operations shall be allowed as described in 7.1.1.8. Attempts by a process in a
81 background process group to write to its controlling terminal shall cause the pro-
82 cess group to be sent a SIGTTOU signal unless one of the following special cases
83 apply: If TOSTOP is not set, or if TOSTOP is set and the process is ignoring or
84 blocking the SIGTTOU signal, the process is allowed to write to the terminal and
85 the SIGTTOU signal is not sent. If TOSTOP is set, and the process group of the
86 writing process is orphaned, and the writing process is not ignoring or blocking
87 SIGTTOU, the *write()* returns -1 with *errno* set to [EIO], and no signal is sent.

88 Certain calls that set terminal parameters are treated in the same fashion as
89 write, except that TOSTOP is ignored; that is, the effect is identical to that of ter-
90 minal writes when TOSTOP is set. See 7.2.

91 7.1.1.5 Input Processing and Reading Data

92 A terminal device associated with a terminal device file may operate in full-
93 duplex mode, so that data may arrive even while output is occurring. Each termi-
94 nal device file has associated with it an *input queue*, into which incoming data is
95 stored by the system before being read by a process. The system may impose a
96 limit, {MAX_INPUT}, on the number of bytes that may be stored in the input
97 queue. The behavior of the system when this limit is exceeded is implementation
98 defined.

99 Two general kinds of input processing are available, determined by whether the
100 terminal device file is in canonical mode or noncanonical mode. These modes are
101 described in 7.1.1.6 and 7.1.1.7. Additionally, input characters are processed
102 according to the *c_iflag* (see 7.1.2.2) and *c_lflag* (see 7.1.2.5) fields. Such process-
103 ing can include *echoing*, which in general means transmitting input characters
104 immediately back to the terminal when they are received from the terminal. This
105 is useful for terminals that can operate in full-duplex mode.

106 The manner in which data is provided to a process reading from a terminal device
107 file is dependent on whether the terminal device file is in canonical or noncanoni-
108 cal mode.

109 Another dependency is whether the O_NONBLOCK flag is set by *open()* or *fcntl()*.
110 If the O_NONBLOCK flag is clear, then the read request shall be blocked until
111 data is available or a signal has been received. If the O_NONBLOCK flag is set,
112 then the read request shall be completed, without blocking, in one of three ways:

- 113 (1) If there is enough data available to satisfy the entire request, the *read()*
114 shall complete successfully and return the number of bytes read.
- 115 (2) If there is not enough data available to satisfy the entire request, the
116 *read()* shall complete successfully, having read as much data as possible,
117 and return the number of bytes it was able to read.
- 118 (3) If there is no data available, the *read()* shall return -1 with *errno* set to
119 [EAGAIN].

120 When data is available depends on whether the input processing mode is canoni-
121 cal or noncanonical. The following subclauses, 7.1.1.6 and 7.1.1.7, describe each
122 of these input processing modes.

7.1.1.6 Canonical Mode Input Processing

In canonical mode input processing, terminal input is processed in units of lines. A line is delimited by a newline ('\n') character, an end-of-file (EOF) character, or an end-of-line (EOL) character. See 7.1.1.9 for more information on EOF and EOL. This means that a read request shall not return until an entire line has been typed or a signal has been received. Also, no matter how many bytes are requested in the read call, at most one line shall be returned. It is not, however, necessary to read a whole line at once; any number of bytes, even one, may be requested in a read without losing information.

If {MAX_CANON} is defined for this terminal device, it is a limit on the number of bytes in a line. The behavior of the system when this limit is exceeded is implementation defined. If {MAX_CANON} is not defined, there is no such limit; see 2.8.5.

Erase and kill processing occur when either of two special characters, the ERASE and KILL characters (see 7.1.1.9), is received. This processing affects data in the input queue that has not yet been delimited by a newline (NL), EOF, or EOL character. This undelimited data makes up the current line. The ERASE character deletes the last character in the current line, if there is any. The KILL character deletes all data in the current line, if there is any. The ERASE and KILL characters have no effect if there is no data in the current line. The ERASE and KILL characters themselves are not placed in the input queue.

7.1.1.7 Noncanonical Mode Input Processing

In noncanonical mode input processing, input bytes are not assembled into lines, and erase and kill processing does not occur. The values of the MIN and TIME members of the *c_cc* array are used to determine how to process the bytes received.

MIN represents the minimum number of bytes that should be received when the *read()* function successfully returns. TIME is a timer of 0,1 second granularity that is used to time out short-term or bursty data transmissions. If MIN is greater than {MAX_INPUT}, the response to the request is undefined. The four possible values for MIN and TIME and their interactions are described below.

7.1.1.7.1 Case A: MIN > 0, TIME > 0

In this case TIME serves as an interbyte timer and is activated after the first byte is received. Since it is an interbyte timer, it is reset after a byte is received. The interaction between MIN and TIME is as follows: as soon as one byte is received, the interbyte timer is started. If MIN bytes are received before the interbyte timer expires (remember that the timer is reset upon receipt of each byte), the read is satisfied. If the timer expires before MIN bytes are received, the characters received to that point are returned to the user. Note that if TIME expires, at least one byte shall be returned because the timer would not have been enabled unless a byte was received. In this case (MIN > 0, TIME > 0), the read shall block until the MIN and TIME mechanisms are activated by the receipt of the first byte or until a signal is received. If data is in the buffer at the time of the *read()*, the result shall be as if data had been received immediately after the *read()*.

167 **7.1.1.7.2 Case B: MIN > 0, TIME = 0**

168 In this case, since the value of TIME is zero, the timer plays no role and only MIN
 169 is significant. A pending read is not satisfied until MIN bytes are received (i.e.,
 170 the pending read shall block until MIN bytes are received) or a signal is received.
 171 A program that uses this case to read record-based terminal I/O may block
 172 indefinitely in the read operation.

173 **7.1.1.7.3 Case C: MIN = 0, TIME > 0**

174 In this case, since MIN = 0, TIME no longer represents an interbyte timer. It now
 175 serves as a read timer that is activated as soon as the *read()* function is pro-
 176 cesses. A read is satisfied as soon as a single byte is received or the read timer
 177 expires. Note that in this case if the timer expires, no bytes shall be returned. If
 178 the timer does not expire, the only way the read can be satisfied is if a byte is
 179 received. In this case, the read shall not block indefinitely waiting for a byte; if no
 180 byte is received within TIME*0,1 seconds after the read is initiated, the *read()*
 181 shall return a value of zero, having read no data. If data is in the buffer at the
 182 time of the *read()*, the timer shall be started as if data had been received immedi-
 183 ately after the *read()*.

184 **7.1.1.7.4 Case D: MIN = 0, TIME = 0**

185 The minimum of either the number of bytes requested or the number of bytes
 186 currently available shall be returned without waiting for more bytes to be input.
 187 If no characters are available, *read()* shall return a value of zero, having read no
 188 data.

189 **7.1.1.8 Writing Data and Output Processing**

190 When a process writes one or more bytes to a terminal device file, they are pro-
 191 cessed according to the *c_oflag* field (see 7.1.2.3). The implementation may pro-
 192 vide a buffering mechanism; as such, when a call to *write()* completes, all of the
 193 bytes written have been scheduled for transmission to the device, but the
 194 transmission will not necessarily have completed. See also 6.4.2 for the effects of
 195 O_NONBLOCK on *write()*.

196 **7.1.1.9 Special Characters**

197 Certain characters have special functions on input or output or both. These func-
 198 tions are summarized as follows:

199	INTR	Special character on input and recognized if the ISIG flag (see
200		7.1.2.5) is enabled. It generates a SIGINT signal that is sent to
201		all processes in the foreground process group for which the ter-
202		terminal is the controlling terminal. If ISIG is set, the INTR char-
203		acter is discarded when processed.
204	QUIT	Special character on input and recognized if the ISIG flag is
205		enabled. It generates a SIGQUIT signal that is sent to all
206		processes in the foreground process group for which the termi-
207		nal is the controlling terminal. If ISIG is set, the QUIT charac-
208		ter is discarded when processed.

9	ERASE	Special character on input and recognized if the ICANON flag is set. It erases the last character in the current line; see 7.1.1.6. The ERASE character shall not erase beyond the start of a line, as delimited by an NL, EOF, or EOL character. If ICANON is set, the ERASE character is discarded when processed.
4	KILL	Special character on input and recognized if the ICANON flag is set. It deletes the entire line, as delimited by a NL, EOF, or EOL character. If ICANON is set, the KILL character is discarded when processed.
8	EOF	Special character on input and recognized if the ICANON flag is set. When received, all the bytes waiting to be read are immediately passed to the process, without waiting for a new-line, 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 <i>read()</i> , representing an end-of-file indication. If ICANON is set, the EOF character is discarded when processed.
6	NL	Special character on input and recognized if the ICANON flag is set. It is the line delimiter ('\n').
8	EOL	Special character on input and recognized if the ICANON flag is set. It is an additional line delimiter, like NL.
0	SUSP	Recognized on input if job control is supported (see 7.1.2.6). If the ISIG flag is enabled, receipt of the SUSP character causes a SIGTSTP signal to be sent to all processes in the foreground process group for which the terminal is the controlling terminal, and the SUSP character is discarded when processed.
5	STOP	Special character on both input and output and recognized if the IXON (output control) or IXOFF (input control) flag is set. It can be used to temporarily suspend output. It is useful with CRT terminals to prevent output from disappearing before it can be read. If IXON is set, the STOP character is discarded when processed.
1	START	Special character on both input and output and recognized if the IXON (output control) or IXOFF (input control) flag is set. Can be used to resume output that has been suspended by a STOP character. If IXON is set, the START character is discarded when processed.
6	CR	Special character on input and recognized if the ICANON flag is set; it is the '\r', as denoted in the C Standard [2]. When ICANON and ICRNL are set and IGNCR is not set, this character is translated into a NL and has the same effect as a NL character.

1 The NL and CR characters cannot be changed. It is implementation defined
2 whether the START and STOP characters can be changed. The values for INTR,
3 QUIT, ERASE, KILL, EOF, EOL, and SUSP (job control only), shall be changeable to
4 suit individual tastes.

255 If `(_POSIX_VDISABLE)` is in effect for the terminal file, special character functions
256 associated with changeable special control characters can be disabled individu-
257 ally; see 7.1.2.6.

258 If two or more special characters have the same value, the function performed
259 when that character is received is undefined.

260 A special character is recognized not only by its value, but also by its context; for
261 example, an implementation may define multibyte sequences that have a mean-
262 ing different from the meaning of the bytes when considered individually. Imple-
263 mentations may also define additional single-byte functions. These
264 implementation-defined multibyte or single-byte functions are recognized only if
265 the `IEXTEN` flag is set; otherwise, data is received without interpretation, except
266 as required to recognize the special characters defined in this subclause (7.1.1.9).

267 7.1.1.10 Modem Disconnect

268 If a modem disconnect is detected by the terminal interface for a controlling ter-
269 minal, and if `CLOCAL` is not set in the `c_cflag` field for the terminal (see 7.1.2.4),
270 the `SIGHUP` signal is sent to the controlling process associated with the terminal.
271 Unless other arrangements have been made, this causes the controlling process to
272 terminate; see 3.2.2. Any subsequent call to the `read()` function shall return the
273 value zero, indicating end of file. See 6.4.1. Thus, processes that read a terminal
274 file and test for end-of-file can terminate appropriately after a disconnect. If the
275 `[EIO]` condition specified in 6.4.1.4 that applies when the implementation supports
276 job control also exists, it is unspecified whether the EOF condition or the `[EIO]` is
277 returned. Any subsequent `write()` to the terminal device returns `-1`, with `errno`
278 set to `[EIO]`, until the device is closed.

279 7.1.1.11 Closing a Terminal Device File

280 The last process to close a terminal device file shall cause any output to be sent to
281 the device and any input to be discarded. Then, if `HUPCL` is set in the control
282 modes and the communications port supports a disconnect function, the terminal
283 device shall perform a disconnect.

284 7.1.2 Parameters That Can Be Set

285 7.1.2.1 *termios* Structure

286 Routines that need to control certain terminal I/O characteristics shall do so by
287 using the *termios* structure as defined in the header `<termios.h>`. The
288 members of this structure include (but are not limited to) those shown in Table 7-
289 1.

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

292

Table 7-1 – *termios* Structure

Member Type	Array Size	Member Name	Description
<i>tcflag_t</i>		<i>c_iflag</i>	Input modes.
<i>tcflag_t</i>		<i>c_oflag</i>	Output modes.
<i>tcflag_t</i>		<i>c_cflag</i>	Control modes.
<i>tcflag_t</i>		<i>c_lflag</i>	Local modes.
<i>cc_t</i>	NCCS	<i>c_cc</i>	Control characters.

7.1.2.2 Input Modes

Values of the *c_iflag* field, shown in Table 7-2, describe the basic terminal input control and are composed of the bitwise inclusive OR of the masks shown, which shall be bitwise distinct. The mask name symbols in this table are defined in `<termios.h>`.

Table 7-2 – *termios* *c_iflag* Field

Mask Name	Description
BRKINT	Signal interrupt on break.
ICRNL	Map CR to NL on input.
IGNBRK	Ignore break condition.
IGNCR	Ignore CR.
IGNPAR	Ignore characters with parity errors.
INLCR	Map NL to CR on input.
INPCK	Enable input parity check.
ISTRIP	Strip character.
IXOFF	Enable start/stop input control.
IXON	Enable start/stop output control.
PARMRK	Mark parity errors.

In the context of asynchronous serial data transmission, a break condition is defined as a sequence of zero-valued bits that continues for more than the time to send one byte. The entire sequence of zero-valued bits is interpreted as a single break condition, even if it continues for a time equivalent to more than one byte. In contexts other than asynchronous serial data transmission, the definition of a break condition is implementation defined.

If IGNBRK is set, a break condition detected on input is ignored, that is, not put on the input queue and therefore not read by any process. If IGNBRK is not set and BRKINT is set, the break condition shall flush the input and output queues. If the terminal is the controlling terminal of a foreground process group, the break condition shall generate a single SIGINT signal to that foreground process group. If neither IGNBRK nor BRKINT is set, a break condition is read as a single `'\0'`, or if PARMRK is set, as `'\377'`, `'\0'`, `'\0'`.

If IGNPAR is set, a byte with a framing or parity error (other than break) is ignored.

338 If PARMRK is set and IGNPAR is not set, a byte with a framing or parity error
 339 (other than break) is given to the application as the three-character sequence
 340 '\377', '\0', X, where '\377', '\0' is a two-character flag preceding each sequence
 341 and X is the data of the character received in error. To avoid ambiguity in this
 342 case, if ISTRIP is not set, a valid character of '\377' is given to the application as
 343 '\377', '\377'. If neither PARMRK nor IGNPAR is set, a framing or parity error
 344 (other than break) is given to the application as a single character '\0'.

345 If INPCK is set, input parity checking is enabled. If INPCK is not set, input parity
 346 checking is disabled, allowing output parity generation without input parity
 347 errors. Note that whether input parity checking is enabled or disabled is
 348 independent of whether parity detection is enabled or disabled (see 7.1.2.4). If
 349 parity detection is enabled, but input parity checking is disabled, the hardware to
 350 which the terminal is connected shall recognize the parity bit, but the terminal
 351 special file shall not check whether this bit is set correctly or not.

352 If ISTRIP is set, valid input bytes are first stripped to seven bits; otherwise, all
 353 eight bits are processed.

354 If INLCR is set, a received NL character is translated into a CR character. If
 355 IGNCR is set, a received CR character is ignored (not read). If IGNCR is not set
 356 and ICRNL is set, a received CR character is translated into a NL character.

357 If IXON is set, start/stop output control is enabled. A received STOP character
 358 shall suspend output, and a received START character shall restart output. When
 359 IXON is set, START and STOP characters are not read, but merely perform flow
 360 control functions. When IXON is not set, the START and STOP characters are
 361 read.

362 If IXOFF is set, start/stop input control is enabled. The system shall transmit one
 363 or more STOP characters, which are intended to cause the terminal device to stop
 364 transmitting data, as needed to prevent the input queue from overflowing and
 365 causing the undefined behavior described in 7.1.1.5 and shall transmit one or
 366 more START characters, which are intended to cause the terminal device to
 367 resume transmitting data, as soon as the device can continue transmitting data
 368 without risk of overflowing the input queue. The precise conditions under which
 369 STOP and START characters are transmitted are implementation defined.

370 The initial input control value after *open()* is implementation defined.

371 7.1.2.3 Output Modes

372 Values of the *c_oflag* field describe the basic terminal output control and are com-
 373 posed of the bitwise inclusive OR of the following masks, which shall be bitwise
 374 distinct:

375	<u>Mask Name</u>	<u>Description</u>
376	OPOST	Perform output processing.

377 The mask name symbols for the *c_oflag* field are defined in `<termios.h>`.

378 If OPOST is set, output data is processed in an implementation-defined fashion so
 379 that lines of text are modified to appear appropriately on the terminal device;

otherwise, characters are transmitted without change.

The initial output control value after *open()* is implementation defined.

7.1.2.4 Control Modes

Values of the *c_cflag* field, shown in Table 7-3, describe the basic terminal hardware control and are composed of the bitwise inclusive OR of the masks shown, which shall be bitwise distinct; not all values specified are required to be supported by the underlying hardware. The mask name symbols in this table are defined in `<termios.h>`.

Table 7-3 – *termios* *c_cflag* Field

Mask Name	Description
CLOCAL	Ignore modem status lines.
CREAD	Enable receiver.
CSIZE	Number of bits per byte:
CS5	5 bits
CS6	6 bits
CS7	7 bits
CS8	8 bits
CSTOPB	Send two stop bits, else one.
HUPCL	Hang up on last close.
PARENB	Parity enable.
PARODD	Odd parity, else even.

The CSIZE bits specify the byte size in bits for both transmission and reception. This size does not include the parity bit, if any. If CSTOPB is set, two stop bits are used; otherwise, one stop bit is used. For example, at 110 baud, two stop bits are normally used.

If CREAD is set, the receiver is enabled; otherwise, no characters shall be received.

If PARENB is set, parity generation and detection is enabled and a parity bit is added to each character. If parity is enabled, PARODD specifies odd parity if set; otherwise, even parity is used.

If HUPCL is set, the modem control lines for the port shall be lowered when the last process with the port open closes the port or the process terminates. The modem connection shall be broken.

If CLOCAL is set, a connection does not depend on the state of the modem status lines. If CLOCAL is clear, the modem status lines shall be monitored.

Under normal circumstances, a call to the *open()* function shall wait for the modem connection to complete. However, if the O_NONBLOCK flag is set (see 5.3.1) or if CLOCAL has been set, the *open()* function shall return immediately without waiting for the connection.

If the object for which the control modes are set is not an asynchronous serial connection, some of the modes may be ignored; for example, if an attempt is made to

423 set the baud rate on a network connection to a terminal on another host, the baud
424 rate may or may not be set on the connection between that terminal and the
425 machine to which it is directly connected.

426 The initial hardware control value after *open()* is implementation defined.

427 7.1.2.5 Local Modes

428 Values of the *c_lflag* field, shown in Table 7-4, describe the control of various func-
429 tions and are composed of the bitwise inclusive OR of the masks shown, which
430 shall be bitwise distinct. The mask name symbols in this table are defined in
431 `<termios.h>`.

432 **Table 7-4 – *termios* *c_lflag* Field**

433	Mask Name	Description
434		
435	ECHO	Enable echo.
436	ECHOE	Echo ERASE as an error-correcting backspace.
437	ECHOK	Echo KILL.
438	ECHONL	Echo '\n'.
439	ICANON	Canonical input (erase and kill processing).
440	IEXTEN	Enable extended (implementation-defined) functions.
441	ISIG	Enable signals.
442	NOFLSH	Disable flush after interrupt, quit, or suspend.
443	TOSTOP	Send SIGTTOU for background output.
444		

445 If ECHO is set, input characters are echoed back to the terminal. If ECHO is not
446 set, input characters are not echoed.

447 If ECHOE and ICANON are set, the ERASE character shall cause the terminal to
448 erase the last character in the current line from the display, if possible. If there is
449 no character to erase, an implementation may echo an indication that this was
450 the case or do nothing.

451 If ECHOK and ICANON are set, the KILL character shall either cause the terminal
452 to erase the line from the display or shall echo the '\n' character after the KILL
453 character.

454 If ECHONL and ICANON are set, the '\n' character shall be echoed even if ECHO
455 is not set.

456 If ICANON is set, canonical processing is enabled. This enables the erase and kill
457 edit functions and the assembly of input characters into lines delimited by NL,
458 EOF, and EOL, as described in 7.1.1.6.

459 If ICANON is not set, read requests are satisfied directly from the input queue. A
460 read shall not be satisfied until at least MIN bytes have been received or the
461 timeout value TIME has expired between bytes. The time value represents tenths
462 of seconds. See 7.1.1.7 for more details.

463 If ISIG is set, each input character is checked against the special control charac-
464 ters INTR, QUIT, and SUSP (job control only). If an input character matches one of
465 these control characters, the function associated with that character is performed.

If ISIG is not set, no checking is done. Thus, these special input functions are possible only if ISIG is set.

If IEXTEN is set, implementation-defined functions shall be recognized from the input data. It is implementation defined how IEXTEN being set interacts with ICANON, ISIG, IXON, or IXOFF. If IEXTEN is not set, then implementation-defined functions shall not be recognized, and the corresponding input characters shall be processed as described for ICANON, ISIG, IXON, and IXOFF.

If NOFLSH is set, the normal flush of the input and output queues associated with the INTR, QUIT, and SUSP (job control only) characters shall not be done.

If TOSTOP is set and the implementation supports job control, the signal SIGTTOU is sent to the process group of a process that tries to write to its controlling terminal if it is not in the foreground process group for that terminal. This signal, by default, stops the members of the process group. Otherwise, the output generated by that process is output to the current output stream. Processes that are blocking or ignoring SIGTTOU signals are excepted and allowed to produce output, and the SIGTTOU signal is not sent.

The initial local control value after *open()* is implementation defined.

7.1.2.6 Special Control Characters

The special control characters values are defined by the array *c_cc*. The subscript name and description for each element in both canonical and noncanonical modes are shown in Table 7-5. The subscript name symbols in this table are defined in `<termios.h>`.

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

Subscript Usage		Description
Canonical Mode	Noncanonical Mode	
VEOF		EOF character
VEOL		EOL character
VERASE		ERASE character
VINTR	VINTR	INTR character
VKILL		KILL character
	VMIN	MIN value
VQUIT	VQUIT	QUIT character
VSUSP	VSUSP	SUSP character
	VTIME	TIME value
VSTART	VSTART	START character
VSTOP	VSTOP	STOP character

The subscript values shall be unique, except that the VMIN and VTIME subscripts may have the same values as the VEOF and VEOL subscripts, respectively.

Implementations that do not support job control may ignore the SUSP character value in the *c_cc* array indexed by the VSUSP subscript.

509 The value of NCCS (the number of elements in the *c_cc* array) is unspecified by
510 this part of ISO/IEC 9945.

511 Implementations that do not support changing the START and STOP characters
512 may ignore the character values in the *c_cc* array indexed by the VSTART and
513 VSTOP subscripts when *tcsetattr()* is called, but shall return the value in use
514 when *tcgetattr()* is called.

515 If `{_POSIX_VDISABLE}` is defined for the terminal device file, and the value of one
516 of the changeable special control characters (see 7.1.1.9) is `{_POSIX_VDISABLE}`,
517 that function shall be disabled, that is, no input data shall be recognized as the
518 disabled special character. If ICANON is not set, the value of `{_POSIX_VDISABLE}`
519 has no special meaning for the VMIN and VTIME entries of the *c_cc* array.

520 The initial values of all control characters are implementation defined.

521 7.1.2.7 Baud Rate Values

522 The baud rate values specified in Table 7-6 can be set into the *termios* structure
523 by the baud rate functions in 7.1.3.

524 **Table 7-6 – *termios* Baud Rate Values**

525	Name	Description	Name	Description
526				
527	B0	Hang up	B600	600 baud
528	B50	50 baud	B1200	1200 baud
529	B75	75 baud	B1800	1800 baud
530	B110	110 baud	B2400	2400 baud
531	B134	134.5 baud	B4800	4800 baud
532	B150	150 baud	B9600	9600 baud
533	B200	200 baud	B19200	19 200 baud
534	B300	300 baud	B38400	38 400 baud
535				

536 7.1.3 Baud Rate Functions

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

538 7.1.3.1 Synopsis

```
539 #include <termios.h>
540 speed_t cfgetospeed(const struct termios *termios_p);
541 int cfsetospeed(struct termios *termios_p, speed_t speed);
542 speed_t cfgetispeed(const struct termios *termios_p);
543 int cfsetispeed(struct termios *termios_p, speed_t speed);
```

7.1.3.2 Description

The following interfaces are provided for getting and setting the values of the input and output baud rates in the *termios* structure. The effects on the terminal device described below do not become effective until the *tcsetattr()* function is successfully called, and not all errors are detected until *tcsetattr()* is called as well.

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

The type *speed_t* shall be defined in `<termios.h>` and shall be an unsigned integral type.

The *termios_p* argument is a pointer to a *termios* structure.

The *cfgetospeed()* function shall return the output baud rate stored in the *termios* structure to which *termios_p* points.

The *cfgetispeed()* function shall return the input baud rate stored in the *termios* structure to which *termios_p* points.

The *cfsetospeed()* function shall set the output baud rate stored in the *termios* structure to which *termios_p* points.

The *cfsetispeed()* function shall set the input baud rate stored in the *termios* structure to which *termios_p* points.

Certain values for speeds that are set in the *termios* structure and passed to *tcsetattr()* have special meanings. These are discussed under *tcsetattr()*.

The *cfgetispeed()* and *cfgetospeed()* functions return exactly the value found in the *termios* data structure, without interpretation.

Both *cfsetispeed()* and *cfsetospeed()* return a value of zero if successful and `-1` to indicate an error. It is unspecified whether these return an error if an unsupported baud rate is set.

7.1.3.3 Returns

See 7.1.3.2.

7.1.3.4 Errors

This part of ISO/IEC 9945 does not specify any error conditions that are required to be detected for the *cfgetispeed()*, *cfgetospeed()*, *cfsetispeed()*, or *cfsetospeed()* functions. Some errors may be detected under conditions that are unspecified by this part of ISO/IEC 9945.

7.1.3.5 Cross-References

tcsetattr(), 7.2.1.

579 **7.2 General Terminal Interface Control Functions**

580 The functions that are used to control the general terminal function are described
 581 in this clause. If the implementation supports job control, unless otherwise noted
 582 for a specific command, these functions are restricted from use by background
 583 processes. Attempts to perform these operations shall cause the process group to
 584 be sent a SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU
 585 signals, the process is allowed to perform the operation and the SIGTTOU signal is
 586 not sent.

587 In all the functions, *fildev* is an open file descriptor. However, the functions affect
 588 the underlying terminal file, not just the open file description associated with the
 589 file descriptor.

590 **7.2.1 Get and Set State**

591 Functions: *tcgetattr()*, *tcsetattr()*

592 **7.2.1.1 Synopsis**

```
593 #include <termios.h>
594 int tcgetattr(int fildev, struct termios *termios_p);
595 int tcsetattr(int fildev, int optional_actions,
596               const struct termios * termios_p);
```

597 **7.2.1.2 Description**

598 The *tcgetattr()* function shall get the parameters associated with the object
 599 referred to by *fildev* and store them in the *termios* structure referenced by
 600 *termios_p*. This function is allowed from a background process; however, the ter-
 601 minal attributes may be subsequently changed by a foreground process. If the
 602 terminal device supports different input and output baud rates, the baud rates
 603 stored in the *termios* structure returned by *tcgetattr()* shall reflect the actual baud
 604 rates, even if they are equal. If differing baud rates are not supported, the rate
 605 returned as the output baud rate shall be the actual baud rate. The rate returned
 606 as the input baud rate shall be either the number zero or the output rate (as one
 607 of the symbolic values). Permitting either behavior is obsolescent.⁴⁾

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

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

612 4) In a future revision of this part of ISO/IEC 9945, a returned value of zero as the input baud rate
 613 when differing baud rates are not supported may no longer be permitted.

- 4 (2) If *optional_actions* is TCSADRAIN, the change shall occur after all output
5 written to *fildes* has been transmitted. This function should be used
6 when changing parameters that affect output.
- 7 (3) If *optional_actions* is TCSAFLUSH, the change shall occur after all output
8 written to the object referred to by *fildes* has been transmitted, and all
9 input that has been received, but not read, shall be discarded before the
10 change is made.

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

3 The zero baud rate, B0, is used to terminate the connection. If B0 is specified as
4 the output baud rate when *tcsetattr()* is called, the modem control lines shall no
5 longer be asserted. Normally, this will disconnect the line.

6 If the input baud rate is equal to the numeral zero in the *termios* structure when
7 *tcsetattr()* is called, the input baud rate will be changed by *tcsetattr()* to the same
8 value as that specified by the value of the output baud rate, exactly as if the input
9 rate had been set to the output rate by *cfsetispeed()*. This usage of zero is
10 obsolescent.

1 The *tcsetattr()* function shall return success if it was able to perform any of the
2 requested actions, even if some of the requested actions could not be performed.
3 It shall set all the attributes that the implementation does support as requested
4 and leave all the attributes not supported by the hardware unchanged. If no part
5 of the request can be honored, it shall return -1 and set *errno* to [EINVAL]. If the
6 input and output baud rates differ and are a combination that is not supported,
7 neither baud rate is changed. A subsequent call to *tcgetattr()* shall return the
8 actual state of the terminal device [reflecting both the changes made and not
9 made in the previous *tcsetattr()* call]. The *tcsetattr()* function shall not change the
10 values in the *termios* structure whether or not it actually accepts them.

1 The *termios* structure may have additional fields not defined by this part of
2 ISO/IEC 9945. The effect of the *tcsetattr()* function is undefined if the value of the
3 *termios* structure pointed to by *termios_p* was not derived from the result of a call
4 to *tcgetattr()* on *fildes*; a Strictly Conforming POSIX.1 Application shall modify
5 only fields and flags defined by this part of ISO/IEC 9945 between the call to
6 *tcgetattr()* and *tcsetattr()*, leaving all other fields and flags unmodified.

7 No actions defined by this part of ISO/IEC 9945, other than a call to *tcsetattr()* or a
8 close of the last file descriptor in the system associated with this terminal device,
9 shall cause any of the terminal attributes defined by this part of ISO/IEC 9945 to
10 change.

1 7.2.1.3 Returns

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

654 **7.2.1.4 Errors**

655 If any of the following conditions occur, the *tcgetattr()* function shall return -1
656 and set *errno* to the corresponding value:

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

658 [ENOTTY] The file associated with *fdes* is not a terminal.

659 If any of the following conditions occur, the *tcsetattr()* function shall return -1 and
660 set *errno* to the corresponding value:

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

662 [EINTR] A signal interrupted the *tcsetattr()* function.

663 [EINVAL] The *optional_actions* argument is not a proper value, or an
664 attempt was made to change an attribute represented in the
665 *termios* structure to an unsupported value.

666 [ENOTTY] The file associated with *fdes* is not a terminal.

667 **7.2.1.5 Cross-References**

668 <termios.h>, 7.1.2.

669 **7.2.2 Line Control Functions**

670 Functions: *tcsendbreak()*, *tcdrain()*, *tcflush()*, *tcflow()*

671 **7.2.2.1 Synopsis**

```
672 #include <termios.h>
673 int tcsendbreak(int fdes, int duration);
674 int tcdrain(int fdes);
675 int tcflush(int fdes, int queue_selector);
676 int tcflow(int fdes, int action);
```

677 **7.2.2.2 Description**

678 If the terminal is using asynchronous serial data transmission, the *tcsendbreak()*
679 function shall cause transmission of a continuous stream of zero-valued bits for a
680 specific duration. If *duration* is zero, it shall cause transmission of zero-valued
681 bits for at least 0,25 seconds and not more than 0,5 seconds. If *duration* is not
682 zero, it shall send zero-valued bits for an implementation-defined period of time.

683 If the terminal is not using asynchronous serial data transmission, it is imple-
684 mentation defined whether the *tcsendbreak()* function sends data to generate a
685 break condition (as defined by the implementation) or returns without taking any
686 action.

687 The *tcdrain()* function shall wait until all output written to the object referred to
688 by *fdes* has been transmitted.

Upon successful completion, the *tcflush()* function shall have discarded any data written to the object referred to by *fildes* but not transmitted, or data received, but not read, depending on the value of *queue_selector*:

- (1) If *queue_selector* is TCIFLUSH, it shall flush data received, but not read.
- (2) If *queue_selector* is TCOFLUSH, it shall flush data written, but not transmitted.
- (3) If *queue_selector* is TCIOFLUSH, it shall flush both data received but not read and data written but not transmitted.

The *tcflow()* function shall suspend transmission or reception of data on the object referred to by *fildes*, depending on the value of *action*:

- (1) If *action* is TCOOFF, it shall suspend output.
- (2) If *action* is TCOON, it shall restart suspended output.
- (3) If *action* is TCIOFF, the system shall transmit a STOP character, which is intended to cause the terminal device to stop transmitting data to the system. (See the description of IXOFF in 7.1.2.2.)
- (4) If *action* is TCION, the system shall transmit a START character, which is intended to cause the terminal device to start transmitting data to the system. (See the description of IXOFF in 7.1.2.2.)

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

The default on the opening of a terminal file is that neither its input nor its output is suspended.

7.2.2.3 Returns

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

7.2.2.4 Errors

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

- | | |
|----------|--|
| [EBADF] | The <i>fildes</i> argument is not a valid file descriptor. |
| [ENOTTY] | The file associated with <i>fildes</i> 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 <i>fildes</i> argument is not a valid file descriptor. |
| [EINTR] | A signal interrupted the <i>tcdrain()</i> function. |
| [ENOTTY] | The file associated with <i>fildes</i> is not a terminal. |

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

- 726 [EBADF] The *fil*des argument is not a valid file descriptor.
- 727 [EINVAL] The *queue_selector* argument is not a proper value.
- 728 [ENOTTY] The file associated with *fil*des is not a terminal.
- 729 If any of the following conditions occur, the *tcflow()* function shall return -1 and
730 set *errno* to the corresponding value:
- 731 [EBADF] The *fil*des argument is not a valid file descriptor.
- 732 [EINVAL] The *action* argument is not a proper value.
- 733 [ENOTTY] The file associated with *fil*des is not a terminal.

734 7.2.2.5 Cross-References

735 <termios.h>, 7.1.2.

736 7.2.3 Get Foreground Process Group ID

737 Function: *tcgetpgrp()*

738 7.2.3.1 Synopsis

739 #include <sys/types.h>
740 pid_t tcgetpgrp(int *fil*des);

741 7.2.3.2 Description

742 If `{_POSIX_JOB_CONTROL}` is defined:

- 743 (1) The *tcgetpgrp()* function shall return the value of the process group ID of
744 the foreground process group associated with the terminal.
- 745 (2) The *tcgetpgrp()* function is allowed from a process that is a member of a
746 background process group; however, the information may be subse-
747 quently changed by a process that is a member of a foreground process
748 group.

749 Otherwise:

750 The implementation shall either support the *tcgetpgrp()* function as
751 described above or the *tcgetpgrp()* call shall fail.

752 7.2.3.3 Returns

753 Upon successful completion, *tcgetpgrp()* returns the process group ID of the fore-
754 ground process group associated with the terminal. If there is no foreground pro-
755 cess group, *tcgetpgrp()* shall return a value greater than 1 that does not match
756 the process group ID of any existing process group. Otherwise, a value of -1 is
757 returned and *errno* is set to indicate the error.

7.2.3.4 Errors

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

- [EBADF] The *fildes* argument is not a valid file descriptor.
- [ENOSYS] The *tcgetpgrp()* function is not supported in this implementation.
- [ENOTTY] The calling process does not have a controlling terminal, or the file is not the controlling terminal.

7.2.3.5 Cross-References

setsid(), 4.3.2; *setpgid()*, 4.3.3; *tcsetpgrp()*, 7.2.4.

7.2.4 Set Foreground Process Group ID

Function: *tcsetpgrp()*

7.2.4.1 Synopsis

```
#include <sys/types.h>
int tcsetpgrp(int fildes, pid_t pgrp_id);
```

7.2.4.2 Description

If `[_POSIX_JOB_CONTROL]` is defined:

If the process has a controlling terminal, the *tcsetpgrp()* function shall set the foreground process group ID associated with the terminal to *pgrp_id*. The file associated with *fildes* must be the controlling terminal of the calling process, and the controlling terminal must be currently associated with the session of the calling process. The value of *pgrp_id* must match a process group ID of a process in the same session as the calling process.

Otherwise:

The implementation shall either support the *tcsetpgrp()* function as described above, or the *tcsetpgrp()* call shall fail.

7.2.4.3 Returns

Upon successful completion, *tcsetpgrp()* returns a value of zero. Otherwise, a value of *-1* is returned and *errno* is set to indicate the error.

7.2.4.4 Errors

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

790	[EBADF]	The <i>fdes</i> argument is not a valid file descriptor.
791	[EINVAL]	The value of the <i>pgrp_id</i> argument is not supported by the
792		implementation.
793	[ENOSYS]	The <i>tcsetpgrp()</i> function is not supported in this
794		implementation.
795	[ENOTTY]	The calling process does not have a controlling terminal, or the
796		file is not the controlling terminal, or the controlling terminal is
797		no longer associated with the session of the calling process.
798	[EPERM]	The value of <i>pgrp_id</i> is a value supported by the implementa-
799		tion, but does not match the process group ID of a process in the
800		same session as the calling process.

THIS PAGE WAS
BLANK IN THE ORIGINAL

Section 8: Language-Specific Services for the C Programming Language

8.1 Referenced C Language Routines

The functions listed below are described in the indicated sections of the C Standard [2]. POSIX.1 with the C Language Binding comprises these functions, the extensions to them described in this clause, and the rest of the requirements stipulated in this part of ISO/IEC 9945. The functions appended with plus signs (+) have requirements beyond those set forth in the C Standard [2]. Any implementation claiming conformance to POSIX.1 with the C Language Binding shall comply with the requirements outlined in this clause, the requirements stipulated in the rest of this part of ISO/IEC 9945, and the requirements in the indicated sections of the C Standard [2].

For requirements concerning conformance to this clause, see 1.3.3 and its subclauses.

4.2 Diagnostics

Functions: assert.

4.3 Character Handling

Functions: isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, tolower, toupper.

4.4 Localization

Functions: setlocale+.

4.5 Mathematics

Functions: acos, asin, atan, atan2, cos, sin, tan, cosh, sinh, tanh, exp, frexp, ldexp, log, log10, modf, pow, sqrt, ceil, fabs, floor, fmod.

4.6 Non-Local Jumps

Functions: setjmp, longjmp.

4.9 Input/Output

Functions: clearerr, fclose, feof, ferror, fflush, fgetc, fgets, fopen, fputc, fputs, fread, freopen, fseek, ftell, fwrite, getc, getchar, gets, perror, printf, fprintf, sprintf, putc, putchar, puts, remove, rename+, rewind, scanf, fscanf, sscanf, setbuf, tmpfile, tmpnam, ungetc.

4.10 General Utilities

Functions: abs, atof, atoi, atol, rand, srand, calloc, free, malloc, realloc, abort+, exit, getenv+, bsearch, qsort.

4.11 String Handling

Functions: strcpy, strncpy, strcat, strncat, strcmp, strncmp, strchr, strcspn, strpbrk, strrchr, strspn, strstr, strtok, strlen.

4.12 Date and Time

Functions: `time`, `asctime`, `ctime+`, `gmtime+`, `localtime+`, `mktime+`, `strftime+`.

Systems conforming to this part of ISO/IEC 9945 shall make no distinction between the “text streams” and the “binary streams” described in the C Standard [2].

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

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

The relationship between a time in seconds since the Epoch used as an argument to `gmtime()` and the `tm` structure (defined in `<time.h>`) is that the result shall be as specified in the expression given in the definition of *seconds since the Epoch* in 2.2.2.77, where the names in the structure and in the expression correspond. If the time zone `UCT0` is in effect, this shall also be true for `localtime()` and `mktime()`.

8.1.1 Extensions to Time Functions

The contents of the environment variable named `TZ` (see 2.6) shall be used by the functions `ctime()`, `localtime()`, `strftime()`, and `mktime()` to override the default time zone. The value of `TZ` has one of the two forms (spaces inserted for clarity):

:characters

or:

std offset dst offset , rule

If `TZ` is of the first format (i.e., if the first character is a colon), the characters following the colon are handled in an implementation-defined manner.

The expanded format (for all `TZs` whose value does not have a colon as the first character) is as follows:

stdoffset[dst[offset][,start[/time],end[/time]]]

Where:

<i>std</i> and <i>dst</i>	Indicates no less than three, nor more than <code>{TZNAME_MAX}</code> , bytes that are the designation for the standard (<i>std</i>) or summer (<i>dst</i>) time zone. Only <i>std</i> is required; if <i>dst</i> is missing, then summer time does not apply in this locale. Upper- and lowercase letters are explicitly allowed. Any characters except a leading colon (:) or digits, the comma (,), the minus (−), the plus (+), and the null character are permitted to appear in these fields, but their meaning is unspecified.
---------------------------	---

74 *offset* Indicates the value one must add to the local time to arrive at
75 Coordinated Universal Time. The *offset* has the form:

76 *hh[:mm[:ss]]*

77 The minutes (*mm*) and seconds (*ss*) are optional. The hour (*hh*)
78 shall be required and may be a single digit. The *offset* following
79 *std* shall be required. If no *offset* follows *dst*, summer time is
80 assumed to be one hour ahead of standard time. One or more
81 digits may be used; the value is always interpreted as a decimal
82 number. The hour shall be between zero and 24, and the
83 minutes (and seconds)—if present—between zero and 59. Use of
84 values outside these ranges causes undefined behavior. If pre-
85 ceded by a “-”, the time zone shall be east of the Prime Meri-
86 dian; otherwise it shall be west (which may be indicated by an
87 optional preceding “+”).

88 *rule* Indicates when to change to and back from summer time. The
89 *rule* has the form:

90 *date/time, date/time*

91 where the first *date* describes when the change from standard to
92 summer time occurs and the second *date* describes when the
93 change back happens. Each *time* field describes when, in
94 current local time, the change to the other time is made.

95 The format of *date* shall be one of the following:

96 J*n* The Julian day *n* ($1 \leq n \leq 365$). Leap days shall not be
97 counted. That is, in all years—including leap years—
98 February 28 is day 59 and March 1 is day 60. It is
99 impossible to explicitly refer to the occasional
100 February 29.

101 *n* The zero-based Julian day ($0 \leq n \leq 365$). Leap days
102 shall be counted, and it is possible to refer to
103 February 29.

104 M*m*.*n*.*d*
105 The *d*th day ($0 \leq d \leq 6$) of week *n* of month *m* of the
106 year ($1 \leq n \leq 5$, $1 \leq m \leq 12$, where week 5 means “the
107 last *d* day in month *m*” which may occur in either the
108 fourth or the fifth week). Week 1 is the first week in
109 which the *d*th day occurs. Day zero is Sunday.

110 The *time* has the same format as *offset* except that no leading
111 sign (“-” or “+”) shall be allowed. The default, if *time* is not
112 given, shall be 02:00:00.

113 Whenever *ctime()*, *strftime()*, *mktime()*, or *localtime()* is called, the time zone
114 names contained in the external variable *tzname* shall be set as if the *tzset()* func-
115 tion had been called.

116 Applications are explicitly allowed to change TZ and have the changed TZ apply
117 to themselves.

8 8.1.2 Extensions to *setlocale()* Function

Function: *setlocale()*

0 8.1.2.1 Synopsis

```
1 #include <locale.h>
2 char *setlocale(int category, const char *locale);
```

3 8.1.2.2 Description

4 The *setlocale()* function sets, changes, or queries the locale of the process accord-
5 ing to the values of the *category* and the *locale* arguments. The possible values for
6 *category* include:

```
7 LC_CTYPE
8 LC_COLLATE
9 LC_TIME
0 LC_NUMERIC
1 LC_MONETARY
2 Implementation-defined additional categories
```

3 For POSIX.1 systems, environment variables are defined that correspond to the
4 named categories above and that have the same spelling.

5 The value LC_ALL for *category* names all of the categories of the locale of the pro-
6 cess; LC_ALL is a special constant, not a category. There is an environment vari-
7 able LC_ALL with the semantics noted below.

3 The *locale* argument is a pointer to a character string that can be an explicit
3 string, a NULL pointer, or a null string.

3 When *locale* is an explicit string, the contents of the string are implementation
1 defined except for the value "C". The value "C" for *locale* specifies the minimal
2 environment for C-language translation. If *setlocale()* is not invoked, the "C"
3 locale shall be the locale of the process. The locale name "POSIX" shall be recog-
4 nized. It shall provide the same semantics as the C locale for those functions
5 defined within this part of ISO/IEC 9945 or by the C Standard (2). Extensions or
3 refinements to the POSIX locale beyond those provided by the C locale may be
7 included in future revisions, and other parts of ISO/IEC 9945 are expected to add
3 to the requirements of the POSIX locale.

3 When *locale* is a NULL pointer the locale of the process is queried according to the
3 value of *category*. The content of the string returned is unspecified.

1 When *locale* is a null string, the *setlocale()* function takes the name of the new
2 locale for the specified category from the environment as determined by the first
3 condition met below:

- 1 (1) If LC_ALL is defined in the environment and is not null, the value of
3 LC_ALL is used.
- 3 (2) If there is a variable defined in the environment with the same name as
7 the category and that is not null, the value specified by that environment
3 variable is used.

159 (3) If **LANG** is defined in the environment and is not null, the value of **LANG**
160 is used.

161 If the resulting value is a supported locale, *setlocale()* sets the specified category
162 of the locale of the process to that value and returns the value specified below. If
163 the value does not name a supported locale (and is not null), *setlocale()* returns a
164 NULL pointer, and the locale of the process is not changed by this function call. If
165 no nonnull environment variable is present to supply a value, it is implementa-
166 tion defined whether *setlocale()* sets the specified category of the locale of the pro-
167 cess to a systemwide default value or to "C" or to "POSIX". The possible actual
168 values of the environment variables are implementation defined and should
169 appear in the system documentation.

170 Setting all of the categories of the locale of the process is similar to successively
171 setting each individual category of the locale of the process, except that all error
172 checking is done before any actions are performed. To set all the categories of the
173 locale of the process, *setlocale()* is invoked as:

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

175 In this case, *setlocale()* first verifies that the values of all the environment vari-
176 ables it needs according to the precedence above indicate supported locales. If the
177 value of any of these environment-variable searches yields a locale that is not sup-
178 ported (and nonnull), the *setlocale()* function returns a NULL pointer and the
179 locale of the process is not changed. If all environment variables name supported
180 locales, *setlocale()* then proceeds as if it had been called for each category, using
181 the appropriate value from the associated environment variable or from the
182 implementation-defined default if there is no such value.

183 8.1.2.3 Returns

184 A successful call to *setlocale()* returns a string that corresponds to the locale set.
185 The string returned is such that "a subsequent call with that string and its associ-
186 ated category will restore that part of the process's locale" (C Standard {2}). The
187 string returned shall not be modified by the process, but may be overwritten by a
188 subsequent call to the *setlocale()* function. This string is not required to be the
189 value of the environment variable used, if one was used.

190 8.2 C Language Input/Output Functions

191 This clause describes input/output functions of the C Standard {2} and their
192 interactions with other functions defined by this part of ISO/IEC 9945.

193 All functions specified in the C Standard {2} as operating on a *file name* shall
194 operate on a *pathname*. All functions specified in the C Standard {2} as creating a
195 file shall do so as if they called the *creat()* function with a value appropriate to the
196 C language function for the *path* argument and a value of

```
197       S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH
```

198 for the *mode* argument.

3 The type *FILE* and the terms *file position indicator* and *stream* are those defined
3 by the C Standard [2].

1 A stream is considered local to a single process. After a *fork()* call, each of the
2 parent and child have distinct streams that share an open file description.

3 8.2.1 Map a Stream Pointer to a File Descriptor

4 Function: *fileno()*

5 8.2.1.1 Synopsis

3 #include <stdio.h>
7 int fileno(FILE *stream);

3 8.2.1.2 Description

3 The *fileno()* function returns the integer file descriptor associated with the *stream*
3 (see 5.3.1).

1 The following symbolic values in the <unistd.h> header (see 2.9) define the file
2 descriptors that shall be associated with the C language *stdin*, *stdout*, and *stderr*
3 when the application is started:

<u>Name</u>	<u>Description</u>	<u>Value</u>
STDIN_FILENO	Standard input value, <i>stdin</i> .	0
STDOUT_FILENO	Standard output value, <i>stdout</i> .	1
STDERR_FILENO	Standard error value, <i>stderr</i> .	2

3 At entry to *main()*, these streams shall be in the same state as if they had just
3 been opened with *fdopen()* called with a mode consistent with that required by
3 the C Standard [2] and the file descriptor described above.

1 8.2.1.3 Returns

3 See 8.2.1.2. If an error occurs, a value of -1 is returned and *errno* is set to indi-
3 cate the error.

1 8.2.1.4 Errors

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

1 8.2.1.5 Cross-References

3 *open()*, 5.3.1.

230 **8.2.2 Open a Stream on a File Descriptor**231 Function: *fdopen()*232 **8.2.2.1 Synopsis**233 `#include <stdio.h>`234 `FILE *fdopen(int fildes, const char *type);` |235 **8.2.2.2 Description**236 The *fdopen()* routine associates a stream with a file descriptor.237 The *type* argument is a character string having one of the following values:

238	"r"	Open for reading.
239	"w"	Open for writing.
240	"a"	Open for writing at end-of-file.
241	"r+"	Open for update (reading and writing).
242	"w+"	Open for update (reading and writing).
243	"a+"	Open for update (reading and writing) at end-of-file.

244 The meaning of these flags is exactly as specified by the C Standard {2} for
 245 *fopen()*, except that "w" and "w+" do not cause truncation of the file. Additional
 246 values for the *type* argument may be defined by an implementation.

247 The application shall ensure that the mode of the stream is allowed by the mode
 248 of the open file. |

249 The file position indicator associated with the new stream is set to the position
 250 indicated by the file offset associated with the file descriptor. The error indicator
 251 and end-of-file indicator for the stream shall be cleared. |

252 |

253 **8.2.2.3 Returns**

254 If successful, the *fdopen()* function returns a pointer to a stream. Otherwise, a
 255 NULL pointer is returned and *errno* is set to indicate the error.

256 **8.2.2.4 Errors**

257 This part of ISO/IEC 9945 does not specify any error conditions that are required
 258 to be detected for the *fdopen()* function. Some errors may be detected under con-
 259 ditions that are unspecified by this part of ISO/IEC 9945. |

260 **8.2.2.5 Cross-References**261 *open()*, 5.3.1; *fopen()* [C Standard {2}].

8.2.3 Interactions of Other *FILE*-Type C Functions

A single open file description can be accessed both through streams and through file descriptors. Either a file descriptor or a stream will be called a *handle* on the open file description to which it refers; an open file description may have several handles.

Handles can be created or destroyed by user action without affecting the underlying open file description. Some of the ways to create them include *fcntl()*, *dup()*, *fdopen()*, *fileno()*, and *fork()* (which duplicates existing ones into new processes). They can be destroyed by at least *fclose()*, *close()*, and the *exec* functions (which close some file descriptors and destroy streams).

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

The result of function calls involving any one handle (the *active handle*) are defined elsewhere in this part of ISO/IEC 9945, but if two or more handles are used, and any one of them is a stream, their actions shall be coordinated as described below. If this is not done, the result is undefined.

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

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

The handles need not be in the same process for these rules to apply. Note that after a *fork()*, two handles exist where one existed before. The application shall assure that, if both handles will ever be accessed, that they will both be in a state where the other could become the active handle first. The application shall prepare for a *fork()* exactly as if it were a change of active handle. [If the only action performed by one of the processes is one of the *exec* functions or *_exit()* {not *exit()*}, the handle is never accessed in that process.]

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

- 307 (a) If it is a file descriptor, no action is required.
- 308 (b) If the only further action to be performed on any handle to this open
309 file description is to close it, no action need be taken.
- 310 (c) If it is a stream that is unbuffered, no action need be taken.
- 311 (d) If it is a stream that is line-buffered and the last character written
312 to the stream was a newline [that is, as if a *putc(' \n')* was the
313 most recent operation on that stream], no action need be taken.
- 314 (e) If it is a stream that is open for writing or append (but not also open
315 for reading), either an *fflush()* shall occur or the stream shall be
316 closed.
- 317 (f) If the stream is open for reading and it is at the end of the file
318 [*feof()* is true], no action need be taken.
- 319 (g) If the stream is open with a mode that allows reading and the
320 underlying open file description refers to a device that is capable of
321 seeking, either an *fflush()* shall occur or the stream shall be closed.
- 322 (h) Otherwise, the result is undefined.
- 323 (2) For the second handle: if any previous active handle has called a func-
324 tion that explicitly changed the file offset, except as required above for
325 the first handle, the application shall perform an *lseek()* or an *fseek()* (as
326 appropriate to the type of the handle) to an appropriate location.
- 327 (3) If the active handle ceases to be accessible before the requirements on the
328 first handle above have been met, the state of the open file description
329 becomes undefined. This might occur, for example, during a *fork()* or an
330 *_exit()*.
- 331 (4) The *exec* functions shall be considered to make inaccessible all streams
332 that are open at the time they are called, independent of what streams or
333 file descriptors may be available to the new process image.
- 334 (5) Implementations shall assure that an application, even one consisting of
335 several processes, shall yield correct results (no data is lost or duplicated
336 when writing, all data is written in order, except as requested by seeks)
337 when the rules above are followed, regardless of the sequence of handles
338 used. If the rules above are not followed, the result is unspecified. When
339 these rules are followed, it is implementation defined whether, and under
340 what conditions, all input is seen exactly once.
- 341 (6) Each function that operates on a stream is said to have zero or more
342 *underlying functions*. This means that the stream function shares cer-
343 tain traits with the underlying functions, but does not require that there
344 be any relation between the implementations of the stream function and
345 its underlying functions.
- 346 (7) Also, in the subclauses below, additional requirements on the standard
347 I/O routines, beyond those in the C Standard [2], are given.

8 **8.2.3.1 *fopen()***

9 The *fopen()* function shall allocate a file descriptor as *open()* does. |

0 The underlying function is *open()*.

1 **8.2.3.2 *fclose()***

2 The *fclose()* function shall perform a *close()* on the file descriptor that is associ- |
3 ated with the *FILE* stream. It shall also mark for update the *st_ctime* and
4 *st_mtime* fields of the underlying file, if the stream was writable, and if buffered
5 data had not been written to the file yet.

6 The underlying functions are *write()* and *close()*.

8 **8.2.3.3 *freopen()***

9 The *freopen()* function has the properties of both *fclose()* and *fopen()*. |

0 **8.2.3.4 *fflush()***

1 The *fflush()* function shall mark for update the *st_ctime* and *st_mtime* fields of the |
2 underlying file if the stream was writable and if buffered data had not been writ-
3 ten to the file yet.

4 The underlying functions are *write()* and *lseek()*. |

3 **8.2.3.5 *fgetc()*, *fgets()*, *fread()*, *getc()*, *getchar()*, *gets()*, *scanf()*, *fscanf()***

7 These functions may mark the *st_atime* field for update. The *st_atime* field shall
3 be marked for update by the first successful execution of one of these functions
3 that returns data not supplied by a prior call to *ungetc()*.

0 The underlying functions are *read()* and *lseek()*.

1 **8.2.3.6 *fputc()*, *fputs()*, *fwrite()*, *putc()*, *putchar()*, *puts()*, *printf()*,
2 *fprintf()*** |

3 The *st_ctime* and *st_mtime* fields of the file shall be marked for update between
1 the successful execution of one of these functions and the next successful comple-
3 tion of a call to either *fflush()* or *fclose()* on the same stream or a call to *exit()* or
3 *abort()*.

7 The underlying functions are *write()* and *lseek()*.

3 If *fwrite()* writes greater than zero bytes, but fewer than requested, the error indi- |
3 cator for the stream shall be set. If the underlying *write()* reports an error, *errno* |
3 shall not be modified by *fwrite()*, and the error indicator for the stream shall be |
1 set. |

382 If the implementation provides the *vprintf()* and *vfprintf()* functions from the C
 383 Standard [2], they also shall meet the constraints specified in this part of
 384 ISO/IEC 9945 for (respectively) *printf()* and *fprintf()*.

385 **8.2.3.7 *fseek()*, *rewind()***

386 These functions shall mark the *st_ctime* and *st_mtime* fields of the file for update
 387 if the stream was writable and if buffered data had not yet been written to the
 388 file.

389 The underlying functions are *lseek()* and *write()*.

390 If the most recent operation, other than *ftell()*, on a given stream is *fflush()*, the
 391 file offset in the underlying open file description shall be adjusted to reflect the
 392 location specified by the *fseek()*.

393 **8.2.3.8 *perror()***

394 The *perror()* function shall mark the file associated with the standard error
 395 stream as having been written (*st_ctime*, *st_mtime* marked for update) at some
 396 time between its successful completion and *exit()*, *abort()*, or the completion of
 397 *fflush()* or *fclose()* on *stderr*.

398 **8.2.3.9 *tmpfile()***

399 The *tmpfile()* function shall allocate a file descriptor as *fopen()* does.

400 **8.2.3.10 *ftell()***

401 The underlying function is *lseek()*. The result of *ftell()* after an *fflush()* shall be
 402 the same as the result before the *fflush()*. If the stream is opened in append mode
 403 or if the O_APPEND flag is set as a consequence of dealing with other handles on
 404 the file, the result of *ftell()* on that stream is unspecified.

405 **8.2.3.11 Error Reporting**

406 If any of the functions above return an error indication, the value of *errno* shall be
 407 set to indicate the error condition. If that error condition is one that this part of
 408 ISO/IEC 9945 specifies to be detected by one of the corresponding underlying func-
 409 tions, the value of *errno* shall be the same as the value specified for the underly-
 410 ing function.

411 **8.2.3.12 *exit()*, *abort()***

412 The *exit()* function shall have the effect of *fclose()* on every open stream, with the
 413 properties of *fclose()* as described above. The *abort()* function shall also have
 414 these effects if the call to *abort()* causes process termination, but shall have no
 415 effect on streams otherwise. The C Standard [2] specifies the conditions where
 416 *abort()* does or does not cause process termination. For the purposes of that
 417 specification, a signal that is blocked shall not be considered caught.

8.2.4 Operations on Files — the *remove()* Function

The *remove()* function shall have the same effect on file times as *unlink()*.

8.3 Other C Language Functions

8.3.1 Nonlocal Jumps

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

8.3.1.1 Synopsis

```
#include <setjmp.h>
int sigsetjmp(sigjmp_buf env, int savemask);
void siglongjmp(sigjmp_buf env, int val);
```

8.3.1.2 Description

The *sigsetjmp()* macro shall comply with the definition of the *setjmp()* macro in the C Standard [2]. If the value of the *savemask* argument is not zero, the *sigsetjmp()* function shall also save the current signal mask of the process (see 3.3.1) as part of the calling environment.

The *siglongjmp()* function shall comply with the definition of the *longjmp()* function in the C Standard [2]. If and only if the *env* argument was initialized by a call to the *sigsetjmp()* function with a nonzero *savemask* argument, the *siglongjmp()* function shall restore the saved signal mask.

8.3.1.3 Cross-References

sigaction(), 3.3.4; <signal.h>, 3.3.1; *sigprocmask()*, 3.3.5; *sigsuspend()*, 3.3.7.

8.3.2 Set Time Zone

Function: *tzset()*

8.3.2.1 Synopsis

```
#include <time.h>
void tzset(void);
```

8.3.2.2 Description

The *tzset()* function uses the value of the environment variable **TZ** to set time conversion information used by *localtime()*, *ctime()*, *strftime()*, and *mktime()*. If **TZ** is absent from the environment, implementation-defined default time-zone information shall be used.

448 The *tzset()* function shall set the external variable *tzname*:
449 extern char *tzname[2] = {"*std*", "*dst*"};
450 where *std* and *dst* are as described in 8.1.1.

THIS PAGE WAS
BLANK IN THE ORIGINAL

Section 9: System Databases

1 **9.1 System Databases**

2 The routines described in this section allow an application to access the two sys-
3 tem databases that are described below.

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

- 5 (1) Group name
- 6 (2) Numerical group ID
- 7 (3) List of all users allowed in the group

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

- 9 (1) User name
- 10 (2) Numerical user ID
- 11 (3) Numerical group ID
- 12 (4) Initial working directory
- 13 (5) Initial user program

14 If the initial user program field is null, the system default is used.

15 If the initial working directory field is null, the interpretation of that field is
16 implementation defined.

17 These databases may contain other fields that are unspecified by this part of
18 ISO/IEC 9945.

9.2 Database Access

9.2.1 Group Database Access

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

9.2.1.1 Synopsis

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

struct group *getgrgid(gid_t gid);
struct group *getgrnam(const char *name);
```

9.2.1.2 Description

The *getgrgid()* and *getgrnam()* routines both return pointers to an object of type *struct group* containing an entry from the group database with a matching *gid* or *name*. This structure, which is defined in *<grp.h>*, includes the members shown in Table 9-1.

Table 9-1 – *group* Structure

Member Type	Member Name	Description
<i>char *</i>	<i>gr_name</i>	The name of the group.
<i>gid_t</i>	<i>gr_gid</i>	The numerical group ID.
<i>char **</i>	<i>gr_mem</i>	A null-terminated vector of pointers to the individual member names.

9.2.1.3 Returns

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

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

9.2.1.4 Errors

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

9.2.1.5 Cross-References

getlogin(), 4.2.4.

9.2.2 User Database Access

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

9.2.2.1 Synopsis

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

struct passwd *getpwuid(uid_t uid);
struct passwd *getpwnam(const char *name);
```

9.2.2.2 Description

The *getpwuid()* and *getpwnam()* functions both return a pointer to an object of type *struct passwd* containing an entry from the user database with a matching *uid* or *name*. This structure, which is defined in *<pwd.h>*, includes the members shown in Table 9-2.

Table 9-2 – *passwd* Structure

Member Type	Member Name	Description
<i>char *</i>	<i>pw_name</i>	User name.
<i>uid_t</i>	<i>pw_uid</i>	User ID number.
<i>gid_t</i>	<i>pw_gid</i>	Group ID number.
<i>char *</i>	<i>pw_dir</i>	Initial Working Directory.
<i>char *</i>	<i>pw_shell</i>	Initial User Program.

9.2.2.3 Returns

A NULL pointer is returned on error or if the requested entry is not found.
The return values may point to static data that is overwritten on each call.

9.2.2.4 Errors

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

9.2.2.5 Cross-References

getlogin(), 4.2.4.

THIS PAGE WAS
BLANK IN THE ORIGINAL

Section 10: Data Interchange Format

1 10.1 Archive/Interchange File Format

2 A conforming system shall provide a mechanism to copy files from a medium to
3 the file hierarchy and copy files from the file hierarchy to a medium using the
4 interchange formats described here. This part of ISO/IEC 9945 does not define
5 this mechanism.

6 When this mechanism is used to copy files from the medium by a process without
7 appropriate privileges, the protection information (ownership and access permis-
8 sions) shall be set in the same fashion that *creat()* would when given the *mode*
9 argument matching the file permissions supplied by the *mode* field of the
10 extended *tar* format or the *c_mode* field of the extended *cpio* format. A process
11 with appropriate privileges shall restore the ownership and the permissions
12 exactly as recorded on the medium, except that the symbolic user and group IDs
13 are used for the *tar* format, as described in 10.1.1.

14 The *format-creating utility* is used to translate from the file system to the formats
15 defined in this clause. The *format-reading utility* is used to translate from the for-
16 mats defined in this clause to a file system. The interface to these utilities,
17 including their name or names, is implementation defined.

18 The headers of these formats are defined to use characters represented in
19 ISO/IEC 646 {1}; however, no restrictions are placed on the contents of the files
20 themselves. The data in a file may be binary data or text represented in any for-
21 mat available to the user. When these formats are used to transfer text at the
22 source level, all characters shall be represented in ISO/IEC 646 {1} International
23 Reference Version (IRV).

24 The media format and the frames on the media in which the data appear are
25 unspecified by this part of ISO/IEC 9945.

26 NOTE: Guidelines are given in Annex B.

27 10.1.1 Extended *tar* Format

28 An extended *tar* archive tape or file contains a series of blocks. Each block is a
29 fixed-size block of 512 bytes (see below). Although this format may be thought of
30 as being stored on 9-track industry-standard 12,7 mm (0,5 in) magnetic tape,
31 other types of transportable media are not excluded. Each file archived is
32 represented by a header block that describes the file, followed by zero or more
33 blocks that give the contents of the file. At the end of the archive file are two
34 blocks filled with binary zeroes, interpreted as an end-of-archive indicator.

The blocks may be grouped for physical I/O operations. Each group of n blocks (where n is set by the application utility creating the archive file) may be written with a single *write()* operation. On magnetic tape, the result of this write is a single tape record. The last group of blocks is always at the full size, so blocks after the two zero blocks contain undefined data.

The header block is structured as shown in Table 10-1. All lengths and offsets are in decimal.

Table 10-1 – tar Header Block

Field Name	Byte Offset	Length (in bytes)
<i>name</i>	0	100
<i>mode</i>	100	8
<i>uid</i>	108	8
<i>gid</i>	116	8
<i>size</i>	124	12
<i>mtime</i>	136	12
<i>chksum</i>	148	8
<i>typeflag</i>	156	1
<i>linkname</i>	157	100
<i>magic</i>	257	6
<i>version</i>	263	2
<i>uname</i>	265	32
<i>gname</i>	297	32
<i>devmajor</i>	329	8
<i>devminor</i>	337	8
<i>prefix</i>	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 TVERSLN   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 */
```

```

83  #define TUREAD    00 400    /* Read by owner */
84  #define TUWRITE   00 200    /* Write by owner */
85  #define TUEXEC    00 100    /* Execute/Search by owner */
86  #define TGREAD    00 040    /* Read by group */
87  #define TGWRITE   00 020    /* Write by group */
88  #define TGEXEC    00 010    /* Execute/Search by group */
89  #define TOREAD    00 004    /* Read by other */
90  #define TOWRITE   00 002    /* Write by other */
91  #define TOEXEC    00 001    /* Execute/Search by other */

```

92 All characters are represented in the coded character set of ISO/IEC 646 {1}. For
 93 maximum portability between implementations, names should be selected from
 94 characters represented by the portable filename character set as 8-bit characters
 95 with most significant bit zero. If an implementation supports the use of charac-
 96 ters outside the portable filename character set in names for files, users, and
 97 groups, one or more implementation-defined encodings of these characters shall
 98 be provided for interchange purposes. However, the format-reading utility shall
 99 never create file names on the local system that cannot be accessed via the func-
 100 tions described previously in this part of ISO/IEC 9945; see 5.3.1, 5.6.2, 5.2.1,
 101 6.5.2, and 5.1.2. If a file name is found on the medium that would create an
 102 invalid file name, the implementation shall define if the data from the file is
 103 stored on the file hierarchy and under what name it is stored. A format-reading
 104 utility may choose to ignore these files as long as it produces an error indicating
 105 that the file is being ignored.

106 Each field within the header block is contiguous; that is, there is no padding used.
 107 Each character on the archive medium is stored contiguously.

108 The fields *magic*, *uname*, and *gname* are null-terminated character strings. The
 109 fields *name*, *linkname*, and *prefix* are null-terminated character strings except
 110 when all characters in the array contain nonnull characters including the last
 111 character. The *version* field is two bytes containing the characters "00" (zero-
 112 zero). The *typeflag* contains a single character. All other fields are leading zero-
 113 filled octal numbers using digits from ISO/IEC 646 {1} IRV. Each numeric field is
 114 terminated by one or more space or null characters.

115 The *name* and the *prefix* fields produce the pathname of the file. The hierarchical
 116 relationship of the file is retained by specifying the pathname as a path prefix,
 117 and a slash character and filename as the suffix. A new pathname is formed, if
 118 *prefix* is not an empty string (its first character is not null), by concatenating
 119 *prefix* (up to the first null character), a slash character, and *name*; otherwise,
 120 *name* is used alone. In either case, *name* is terminated at the first null character.
 121 If *prefix* is an empty string, it is simply ignored. In this manner, pathnames of at
 122 most 256 characters can be supported. If a pathname does not fit in the space
 123 provided, the format-creating utility shall notify the user of the error, and no
 124 attempt shall be made by the format-creating utility to store any part of the file—
 125 header or data—on the medium.

126 The *linkname* field, described below, does not use the *prefix* to produce a path-
 127 name. As such, a *linkname* is limited to 100 characters. If the name does not fit
 128 in the space provided, the format-creating utility shall notify the user of the error,
 129 and the utility shall not attempt to store the link on the medium.

The *mode* field provides 9 bits specifying file permissions and 3 bits to specify the set UID, set GID, and TSVTX modes. Values for these bits were defined previously. When appropriate privilege is required to set one of these mode bits, and the user restoring the files from the archive does not have the appropriate privilege, the mode bits for which the user does not have appropriate privilege shall be ignored. Some of the mode bits in the archive format are not mentioned elsewhere in this part of ISO/IEC 9945. If the implementation does not support those bits, they may be ignored.

The *uid* and *gid* fields are the user and group ID of the owner and group of the file, respectively.

The *size* field is the size of the file in bytes. If the *typeflag* field is set to specify a file to be of type LNKTYPE or SYMTYPE, the *size* field shall be specified as zero. If the *typeflag* field is set to specify a file of type DIRTYPE, the *size* field is interpreted as described under the definition of that record type. No data blocks are stored for LNKTYPE, SYMTYPE, or DIRTYPE. If the *typeflag* field is set to CHRTYPE, BLKTYPE, or FIFOTYPE, the meaning of the *size* field is unspecified by this part of ISO/IEC 9945, and no data blocks are stored on the medium. Additionally, for FIFOTYPE, the *size* field shall be ignored when reading. If the *typeflag* field is set to any other value, the number of blocks written following the header is $(size+511)/512$, ignoring any fraction in the result of the division.

The *mtime* field is the modification time of the file at the time it was archived. It is the ISO/IEC 646 {1} representation of the octal value of the modification time obtained from the *stat()* function.

The *chksum* field is the ISO/IEC 646 {1} IRV representation of the octal value of the simple sum of all bytes in the header block. Each 8-bit byte in the header is treated as an unsigned value. These values are added to an unsigned integer, initialized to zero, the precision of which shall be no less than 17 bits. When calculating the checksum, the *chksum* field is treated as if it were all blanks.

The *typeflag* field specifies the type of file archived. If a particular implementation does not recognize the type, or the user does not have appropriate privilege to create that type, the file shall be extracted as if it were a regular file if the file type is defined to have a meaning for the size field that could cause data blocks to be written on the medium (see the previous description for *size*). If conversion to an ordinary file occurs, the format-reading utility shall produce an error indicating that the conversion took place. All of the *typeflag* fields are coded in ISO/IEC 646 {1} IRV:

'0'	Represents a regular file. For backward compatibility, a <i>typeflag</i> 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 <i>typeflag</i> value of ISO/IEC 646 {1} IRV '0'.
'1'	Represents a file linked to another file, of any type, previously archived. Such files are identified by each file having the same device and file serial number. The linked-to name is specified in the <i>linkname</i> field with a null terminator if it is less than 100 bytes in length.

176 ' 2' Reserved to represent a link to another file, of any type, whose device or file serial number differs. This is provided for systems that
177 support linked files whose device or file serial numbers differ, and
178 should be treated as a type ' 1' file if this extension does not exist.
179

180 ' 3', ' 4' Represent character special files and block special files respectively.
181 In this case the *devmajor* and *devminor* fields shall contain information defining the device, the format of which is unspecified by this
182 part of ISO/IEC 9945. Implementations may map the device
183 specifications to their own local specification or may ignore the
184 entry.
185

186 ' 5' Specifies a directory or subdirectory. On systems where disk allocation is performed on a directory basis, the *size* field shall contain the
187 maximum number of bytes (which may be rounded to the nearest
188 disk block allocation unit) that the directory may hold. A *size* field
189 of zero indicates no such limiting. Systems that do not support limiting in this manner should ignore the *size* field.
190

192 ' 6' Specifies a FIFO special file. Note that the archiving of a FIFO file archives the existence of this file and not its contents.
193

194 ' 7' Reserved to represent a file to which an implementation has associated some high performance attribute. Implementations without
195 such extensions should treat this file as a regular file (type ' 0').
196

197 ' A' - ' Z' The letters A through Z are reserved for custom implementations.
198 All other values are reserved for specification in future revisions of
199 this part of ISO/IEC 9945.

200 The *magic* field is the specification that this archive was output in this archive
201 format. If this field contains TMAGIC, the *uname* and *gname* fields shall contain
202 the ISO/IEC 646 (1) IRV representation of the owner and group of the file respectively (truncated to fit, if necessary). When the file is restored by a privileged,
203 protection-preserving version of the utility, the password and group files shall be
204 scanned for these names. If found, the user and group IDs contained within these
205 files shall be used rather than the values contained within the *uid* and *gid* fields.
206

207 The encoding of the header is designed to be portable across machines.

208 10.1.1.1 Cross-References

209 <grp.h>, 9.2.1; <pwd.h>, 9.2.2; <sys/stat.h>, 5.6.1; *stat()*, 5.6.2;
210 <unistd.h>, 2.9.

211 10.1.2 Extended *cpio* Format

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

215 An archive may be recorded as a series of fixed-size blocks of bytes. This blocking
216 shall be used only to make physical I/O more efficient. The last group of blocks is
217 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
<i>c_magic</i>	6	Octal number
<i>c_dev</i>	6	Octal number
<i>c_ino</i>	6	Octal number
<i>c_mode</i>	6	Octal number
<i>c_uid</i>	6	Octal number
<i>c_gid</i>	6	Octal number
<i>c_nlink</i>	6	Octal number
<i>c_rdev</i>	6	Octal number
<i>c_mtime</i>	11	Octal number
<i>c_namesize</i>	6	Octal number
<i>c_filesize</i>	11	Octal number

File Name		
Field Name	Length	Interpreted as
<i>c_name</i>	<i>c_namesize</i>	Pathname string

File Data		
Field Name	Length	Interpreted as
<i>c_filedata</i>	<i>c_filesize</i>	Data

10.1.2.1 `cpio` Header

For each file in the archive, a header as defined previously shall be written. The information in the header fields shall be written as streams of ISO/IEC 646 (1) characters interpreted as octal numbers. The octal numbers are extended to the necessary length by appending ISO/IEC 646 (1) IRV zeros at the most-significant-digit end of the number; the result is written to the stream of bytes most-significant-digit first. The fields shall be interpreted as follows:

- (1) *c_magic* shall identify the archive as being a transportable archive by containing the magic bytes as defined by MAGIC (070707).
- (2) *c_dev* and *c_ino* shall contain values that uniquely identify the file within the archive (i.e., no files shall contain the same pair of *c_dev* and *c_ino* values unless they are links to the same file). The values shall be determined in an unspecified manner.
- (3) *c_mode* shall contain the file type and access permissions as defined in Table 10-3.
- (4) *c_uid* shall contain the user ID of the owner.
- (5) *c_gid* shall contain the group ID of the group.

Table 10-3 – Values for *cpio c_mode* Field

259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274

275
276
277
278
279
280
281
282
283
284
285

Table 10-3 – Values for <i>cpio c_mode</i> Field			
File Permissions			
	Name	Value	Indicates
	C_IRUSR	000 400	Read by owner.
	C_IWUSR	000 200	Write by owner.
	C_IXUSR	000 100	Execute by owner.
	C_IRGRP	000 040	Read by group.
	C_IWGRP	000 020	Write by group.
	C_IXGRP	000 010	Execute by group.
	C_IROTH	000 004	Read by others.
	C_IWOTH	000 002	Write by others.
	C_IXOTH	000 001	Execute by others.
	C_ISUID	004 000	Set <i>uid</i> .
	C_ISGID	002 000	Set <i>gid</i> .
	C_ISVTX	001 000	Reserved.
File Type			
	Name	Value	Indicates
	C_ISDIR	040 000	Directory.
	C_ISFIFO	010 000	FIFO.
	C_ISREG	0100 000	Regular file.
	C_ISBLK	060 000	Block special file.
	C_ISCHR	020 000	Character special file.
	C_ISCTG	0110 000	Reserved.
	C_ISLNK	0120 000	Reserved.
	C_ISSOCK	0140 000	Reserved.

286
287
288
289
290
291
292
293
294
295

(6) *c_nlink* shall contain the number of links referencing the file at the time the archive was created.

(7) *c_rdev* shall contain implementation-defined information for character or block special files.

(8) *c_mtime* shall contain the latest time of modification of the file at the time the archive was created.

(9) *c_namesize* shall contain the length of the pathname, including the terminating null byte.

(10) *c_filesize* shall contain the length of the file in bytes. This is the length of the data section following the header structure.

10.1.2.2 *cpio* File Name

c_name shall contain the pathname of the file. The length of this field in bytes is the value of *c_namesize*. If a file name is found on the medium that would create an invalid pathname, the implementation shall define if the data from the file is stored on the file hierarchy and under what name it is stored.

All characters are represented in ISO/IEC 646 {1} IRV. For maximum portability between implementations, names should be selected from characters represented by the portable filename character set as 8-bit characters most significant bit zero. If an implementation supports the use of characters outside the portable filename character set in names for files, users, and groups, one or more implementation-defined encodings of these characters shall be provided for interchange purposes. However, the format-reading utility shall never create file names on the local system that cannot be accessed via the functions described previously in this part of ISO/IEC 9945; see *open()*, *stat()*, *chdir()*, *fcntl()*, and *opendir()*. If a file name is found on the medium that would create an invalid file name, the implementation shall define if the data from the file is stored on the local file system and under what name it is stored. A format-reading utility may choose to ignore these files as long as it produces an error indicating that the file is being ignored.

10.1.2.3 *cpio* File Data

Following *c_name*, there shall be *c_filesize* bytes of data. Interpretation of such data shall occur in a manner dependent on the file. If *c_filesize* is zero, no data shall be contained in *c_filedata*.

10.1.2.4 *cpio* Special Entries

FIFO special files, directories, and the trailer are recorded with *c_filesize* equal to zero. For other special files, *c_filesize* is unspecified by this part of ISO/IEC 9945. The header for the next file entry in the archive shall be written directly after the last byte of the file entry preceding it. A header denoting the file name "TRAILER!!!" shall indicate the end of the archive; the contents of bytes in the last block of the archive following such a header are undefined.

10.1.2.5 *cpio* Values

Values needed by the *cpio* archive format are described in Table 10-3.

C_ISDIR, C_ISFIFO, and C_ISREG shall be supported on a system conforming to this part of ISO/IEC 9945; additional values defined previously are reserved for compatibility with existing systems. Additional file types may be supported; however, such files should not be written on archives intended for transport to portable systems.

C_ISVTX, C_ISCTG, C_ISLNK, and C_ISSOCK have been reserved by this part of ISO/IEC 9945 to retain compatibility with some existing implementations.

When restoring from an archive:

- (1) If the user does not have the appropriate privilege to create a file of the specified type, the format-interpreting utility shall ignore the entry and issue an error to the standard error output.
- (2) Only regular files have data to be restored. Presuming a regular file meets any selection criteria that might be imposed on the format-reading utility by the user, such data shall be restored.

- 341 (3) If a user does not have appropriate privilege to set a particular mode flag,
342 the flag shall be ignored. Some of the mode flags in the archive format
343 are not mentioned elsewhere in this part of ISO/IEC 9945. If the imple-
344 mentation does not support those flags, they may be ignored.

345 **10.1.2.6 Cross-References**

346 <grp.h>, 9.2.1; <pwd.h>, 9.2.2; <sys/stat.h>, 5.6.1; *chmod()*, 5.6.4; *link()*,
347 5.3.4; *mkdir()*, 5.4.1; *read()*, 6.4.1; *stat()*, 5.6.2.

348 **10.1.3 Multiple Volumes**

349 It shall be possible for data represented by the Archive/Interchange File Format
350 to reside in more than one file.

351 The format is considered a stream of bytes. An end-of-file (or equivalently an
352 end-of-media) condition may occur between any two bytes of the logical byte
353 stream. If this condition occurs, the byte following the end-of-file will be the first
354 byte on the next file. The format-reading utility shall, in an implementation-
355 defined manner, determine what file to read as the next file.

THIS PAGE WAS
BLANK IN THE ORIGINAL

Annex A (informative) Bibliography

1 This Annex contains lists of related open systems standards and suggested read-
2 ing on historical implementations and application programming.

3 A.1 Related Open Systems Standards

4 A.1.1 Networking Standards

- 5 {B1} ISO 7498: 1984, *Information processing systems—Open Systems Inter-*
6 *connection—Basic Reference Model.*¹⁾
- 7 {B2} ISO 8072: 1986, *Information processing systems—Open Systems Inter-*
8 *connection—Transport service definition.*
- 9 {B3} ISO/IEC 8073: 1988, *Information processing systems—Open Systems Inter-*
10 *connection—Connection oriented transport protocol specification.*²⁾
- 11 {B4} ISO 8326: 1987, *Information processing systems—Open Systems Inter-*
12 *connection—Basic connection oriented session service definition.*
- 13 {B5} ISO 8327: 1987, *Information processing systems—Open Systems Inter-*
14 *connection—Basic connection oriented session protocol definition.*
- 15 {B6} ISO 8348: 1987, *Information processing systems—Data communications—*
16 *Network service definition.*
- 17 {B7} ISO 8473: 1988, *Information processing systems—Data communications—*
18 *Protocol for providing the connectionless-mode network service.*
- 19 {B8} ISO 8571: 1988, *Information processing systems—Open Systems Inter-*
20 *connection—File Transfer, Access and Management.*

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

23 2) IEC documents can be obtained from the IEC office, 3, rue de Varembe, Case Postale 131, CH-
24 1211, Genève 20, Switzerland/Suisse.

- 25 {B9} ISO 8649: 1988, *Information processing systems—Open Systems Inter-*
26 *connection—Service definition for the Association Control Service Element.*
- 27 {B10} ISO 8650: 1988, *Information processing systems—Open Systems Inter-*
28 *connection—Protocol specification for the Association Control Service Ele-*
29 *ment.*
- 30 {B11} ISO 8802-2: 1989 [IEEE Std 802.2-1989 (ANSI)], *Information processing*
31 *systems—Local area networks—Part 2: Logical link control.*
- 32 {B12} ISO 8802-3: 1989 [IEEE Std 802.3-1988 (ANSI)], *Information processing*
33 *systems—Local area networks—Part 3: Carrier sense multiple access with*
34 *collision detection (CSMA/CD) access method and physical layer*
35 *specifications.*
- 36 {B13} ISO/IEC 8802-4: 1990 [IEEE Std 802.4-1990 (ANSI)], *Information*
37 *technology—Local area networks—Part 4: Token-passing bus access method*
38 *and physical layer specifications.*
- 39 {B14} ISO 8802-5: ... (IEEE 802.5-1989), *Information technology—Local area*
40 *networks—Part 5: Token ring access method and physical layer*
41 *specifications.*
- 42 {B15} ISO 8822: 1988, *Information processing systems—Open Systems Inter-*
43 *connection—Connection oriented presentation service definition.*
- 44 {B16} ISO 8823: 1988, *Information processing systems—Open Systems Inter-*
45 *connection—Connection oriented presentation protocol specification.*
- 46 {B17} ISO 8831: 1989, *Information processing systems—Open Systems Inter-*
47 *connection—Job transfer and manipulation concepts and services.*
- 48 {B18} ISO 8832: 1989, *Information processing systems—Open Systems Inter-*
49 *connection—Specification of the basic class protocol for job transfer and*
50 *manipulation.*
- 51 {B19} CCITT Recommendation X.25, *Interface between data terminal equipment*
52 *(DTE) and data circuit-terminating equipment (DCT) for terminals operating*
53 *in the packet mode and connected to public data networks by dedicated cir-*
54 *cuit.*³⁾
- 55 {B20} CCITT Recommendation X.212, *Information processing systems—Data*
56 *communication—Data link service definition for Open Systems Interconnec-*
57 *tion.*

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

61 **A.1.2 Language Standards**62 {B21} ISO 1539: 1980, *Programming languages—FORTRAN.*63 {B22} ISO 1989: 1985, *Programming Languages—COBOL.*64 {B23} ISO 8652: 1987, *Programming Languages—Ada.*65 {B24} ANSI X3.113-1987⁴⁾, *Information systems—Programming language—FULL*
66 *BASIC.*67 {B25} ANSI/IEEE 770X3.97-1983, *Standard Pascal Computer Programming*
68 *Language.*69 {B26} ANSI/MDC X11.1-1984, *Programming Language MUMPS.*70 **A.1.3 Graphics Standards**71 {B27} ISO 7942: 1985, *Information processing systems—Computer graphics—*
72 *Graphical Kernel System (GKS) functional description.*73 {B28} ISO 8632: 1987, *Information processing systems—Computer graphics—*
74 *Metafile for the storage and transfer of picture description information.*75 {B29} ISO/IEC 9592: 1989 (ANSI X3.144-1988), *Information processing systems—*
76 *Computer graphics—Programmer's hierarchical interactive graphics system*
77 *(PHIGS).*78 **A.1.4 Database Standards**79 {B30} ISO 8907: 1987, *Database Language—NDL.*80 {B31} ISO 9075: 1987, *Database Language—SQL.*81 **A.2 Other Standards**82 {B32} ISO 639: 1988, *Code for the representation of names of languages.*83 {B33} ISO 3166: 1988, *Code for the representation of names of countries.*84 {B34} ISO 8859-1: 1987, *Information Processing—8-bit single-byte coded graphic*
85 *character sets—Part 1: Latin alphabet No. 1.*86 {B35} ISO 9127: 1988, *Information processing systems—User documentation and*
87 *cover information for consumer software packages.*88 {B36} ISO/IEC 9945-2: ..., ⁵⁾ *Information technology—Portable operating system*
89 *interface (POSIX)—Part 2: Shell and utilities.*90 4) ANSI documents can be obtained from the Sales Department, American National Standards
91 Institute, 1430 Broadway, New York, NY 10018.

92 5) To be approved and published.

- [B37] ISO/IEC 10646: ..., ⁶⁾ *Information processing—Multiple octet coded character set.*
- [B38] IEEE Std 100-1988, *IEEE Standard Dictionary of Electrical and Electronics Terms.*

A.3 Historical Documentation and Introductory Texts

- [B39] American Telephone and Telegraph Company. *System V Interface Definition (SVID), Issues 2 and 3.* Morristown, NJ: UNIX Press, 1986, 1989.⁷⁾
- [B40] American Telephone and Telegraph Company. *UNIX System III Programmer's Manual.* Greensboro, NC: Western Electric Company, October 1981.
- [B41] American Telephone and Telegraph Company. *UNIX Time Sharing System: UNIX Programmer's Manual.* 7th ed. Murray Hill, NJ: Bell Telephone Laboratories, January 1979.
- [B42] "The UNIX System."⁸⁾ *AT&T Bell Laboratories Technical Journal.* vol. 63 (8 Part 2), October 1984.
- [B43] "UNIX Time-Sharing System."⁹⁾ *Bell System Technical Journal.* vol. 57 (6 Part 2), July-August 1978.
- [B44] Bach, Maurice J. *The Design of the UNIX Operating System.* Englewood Cliffs, NJ: Prentice-Hall, 1987.
- [B45] Harbison, Samuel P. and Steele, Guy L. C: *A Reference Manual.* Englewood Cliffs, NJ: Prentice-Hall, 1987.
- [B46] Kernighan, Brian W. and Ritchie, Dennis M. *The C Programming Language.* Englewood Cliffs, NJ: Prentice-Hall, 1978.
- [B47] Kernighan, Brian W. and Pike, Rob. *The UNIX Programming Environment.* Englewood Cliffs, NJ: Prentice-Hall, 1984.
- [B48] Leffler, Samuel J., McKusick, Marshall Kirk, Karels, Michael J., Quarterman, John S., and Stettner, Armando. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Reading, MA: Addison-Wesley, 1988.
- [B49] McGilton, Henry and Morgan, Rachel. *Introducing the UNIX System.* New York: McGraw-Hill (BYTE Books), 1983.

⁶⁾ To be approved and published.

⁷⁾ This is one of several documents that represent an industry specification in an area related to POSIX.1. The creators of such documents may be able to identify newer versions that may be interesting.

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

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

- 130 {B50} Organick, Elliot I. *The Multics System: An Examination of Its Structure.* |
131 Cambridge, MA: The MIT Press, 1972. |
- 132 {B51} Quarterman, John S., Silberschatz, Abraham, and Peterson, James L. |
133 "4.2BSD and 4.3BSD as Examples of the UNIX System." *ACM Computing* |
134 *Surveys*. vol. 17 (4), December 1985, pp. 379–418. |
- 135 {B52} Ritchie, Dennis M. "Reflections on Software Research." *Communications* |
136 *of the ACM*. vol. 27 (8), August 1984, pp. 758–760. ACM Turing Award Lec- |
137 ture. |
- 138 {B53} Ritchie, Dennis. "The Evolution of the UNIX Time-Sharing System." *AT&T* |
139 *Bell Laboratories Technical Journal*. vol. 63 (8), October 1984, pp. |
140 1577–1593. |
- 141 {B54} Ritchie, D. M. and Thompson, K. "The UNIX Time-Sharing System." *Com-* |
142 *munications of the ACM*. vol. 7 (7), July 1974, pp. 365–375. This is the ori- |
143 ginal paper, which describes Version 6. |
- 144 {B55} Ritchie, D. M. and Thompson, K. "The UNIX Time-Sharing System." *Bell* |
145 *System Technical Journal*. vol. 57 (6 Part 2), July-August 1978, pp. |
146 1905–1929. This is a revised version and describes Version 7. |
- 147 {B56} Ritchie, Dennis M. "Unix: A Dialectic." *Winter 1987 USENIX Association* |
148 *Conference Proceedings, Washington, D.C.*, pp. 29–34. Berkeley, CA: |
149 USENIX Association, January 1987. |
- 150 {B57} Rochkind, Marc J. *Advanced UNIX Programming*. Englewood Cliffs, NJ: |
151 Prentice-Hall, 1985. |
- 152 {B58} University of California at Berkeley—Computer Science Research Group. |
153 *4.3 Berkeley Software Distribution, Virtual VAX-11 Version*. Berkeley, CA: |
154 The Regents of the University of California, April 1986. |
- 155 {B59} /usr/group Standards Committee. *1984 /usr/group Standard*. Santa |
156 Clara, CA: UniForum, 1984. |
- 157 {B60} X/Open Company, Ltd. *X/Open Portability Guide, Issue 2*. Amsterdam: |
158 Elsevier Science Publishers, 1987. |
- 159 {B61} X/Open Company, Ltd. *X/Open Portability Guide, Issue 3*. Englewood |
160 Cliffs, NJ: Prentice-Hall, 1989. |

THIS PAGE WAS
BLANK IN THE ORIGINAL

Annex B (informative)

Rationale and Notes

The Annex is being published as an informative part of POSIX.1 to assist in the process of review. It contains historical information concerning the contents of POSIX.1 and why features were included or discarded. It also contains notes of interest to application programmers on recommended programming practices, emphasizing the consequences of some aspects of POSIX.1 that may not be immediately apparent.¹⁾

B.1 Scope and Normative References

B.1.1 Scope

This Rationale focuses primarily on additions, clarifications, and changes made to the UNIX system, from which POSIX.1 was derived. It is not a rationale for the UNIX system as a whole, since the goal of POSIX.1's developers was to codify existing practice, not design a new operating system. No attempt is made in this Rationale to defend the pre-existing structure of UNIX systems. It is primarily deviations from existing practice, as codified in the base documents, that are explained or justified here.

Material that is "outside the scope" or otherwise not addressed by this part of ISO/IEC 9945 is implicitly "unspecified." It may be included in an implementation, and thus the implementation does provide a specification for it. The term "implementation-defined" has a specific meaning in POSIX.1 and is not a synonym for "defined (or specified) by the implementation."

The Rationale discusses some UNIX system features that were *not* adopted into POSIX.1. Many of these are features that are popular in some UNIX system implementations, so that a user of those implementations might question why they do not appear in POSIX.1. This Rationale should provide the appropriate answers.

¹⁾ The material in this annex is derived in part from copyrighted draft documents developed under the sponsorship of UniForum, as part of an ongoing program of that association to support the POSIX standards program efforts.

There are choices allowed by POSIX.1 for some details of the interface specification; some of these are specifiable optional subsets of POSIX.1. See B.2.9.

Although the services POSIX.1 provides have been defined in the C language, the concept of providing fundamental, standardized services should not be restricted only to programs of a particular programming language. The possibility of implementing interfaces in alternate programming languages inspired the term *POSIX.1 with the C Language Binding*. The word *Binding* refers to the binding of a conceptual set of services and a standardized C interface that establishes rules and syntax for accessing them. Future international standards are expected to separate the C language binding from the language-independent services of POSIX.1 and to include bindings for other programming languages.

The C Standard [2] will be the basis for functional definitions of core services that are independent of programming languages. POSIX.1 as it stands now can be thought of as a C Language Binding. Sections 1 through 7, and 9, correspond roughly to the C language implementation of what will be defined in the programming language-independent core services portion of POSIX.1; Section 8 corresponds to the C language-specific portion.

The criteria used to choose the programming language-independent core services may be different from those expected. The core services represent services that are common to those programming languages likely to form language bindings to POSIX.1—the greatest common denominator. They are not chosen to reflect the most important system services of an ideal operating system. For this reason, some fundamental system services are not included in the language-independent core. As an example, memory management routines would at first seem to be a core service—they are an absolutely fundamental system service. They must, however, be included in language-specific portions of POSIX.1 because programming languages such as FORTRAN have traditionally not provided memory management. Categorizing memory management as a core service would impose unreasonable requirements for FORTRAN implementations.

Any programming language traditionally supporting memory management should include those routines in the language-dependent portions of their bindings. Work will be done at a later time to standardize the classes of functions that must be included in the language-dependent portions of language bindings if those functions have been traditionally implemented for that language. This will ensure that certain classes of critical functions, such as memory management, will not be excluded from any applicable language binding; see B.1.3.3.

POSIX.1 is not a tutorial on the use of the specified interface, nor is this Rationale. However, the Rationale includes some references to well-regarded historical documentation on the UNIX System in A.3.

B.1.1.1 POSIX.1 and the C Standard

Some C language functions and definitions were handled by POSIX.1, but most were handled by the C Standard [2]. The general guideline is that POSIX.1 retained responsibility for operating-system specific functions, while the C Standard [2] defined C library functions. See also B.2.7 and B.8.

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

- (1) *Function parameter type lists.* These appear in the syntax of the C Standard {2}. In this version of POSIX.1, the parameter lists were res-tated in terms of these function prototypes. There were two major rea-sons for making this change from IEEE Std 1003.1-1988: the use of the C Standard {2} was rapidly becoming more widespread, and implemen-tors were experiencing difficulties with some of the function prototypes where guidance was not provided in POSIX.1. (The modifier *const* pro-vided the most difficulty.) Specific guidance and permission remains in POSIX.1 for translation to common-usage C.
- (2) *Single vs. multiple processes.* The C Standard {2} specifies a language that can be used on single-process operating systems and as a freestand-ing base for the implementation of operating systems or other stand-alone programs. However, the POSIX.1 interface is that of a multiprocess timesharing system. Thus, POSIX.1 has to take multiple processes into account in places where the C Standard {2} does not mention processes at all, such as *kill()*. See also B.1.3.1.1.
- (3) *Single vs. multiple operating system environments.* The C Standard {2} specifies a language that may be useful on more than one operating sys-tem and thus has means of tailoring itself to the particular current environment. POSIX.1 is an operating system interface specification and thus by definition is only concerned with one operating system environ-ment, even though it has been carefully written to be broadly implement-able (see Broadly Implementable in the Introduction) in terms of various underlying operating systems. See also B.1.3.1.1.
- (4) *Translation vs. execution environment.* POSIX.1 is primarily concerned with the C Standard {2} *execution environment*, leaving the *translation environment* to the C Standard {2}. See also B.1.3.1.1.
- (5) *Hosted vs. freestanding implementations.* All POSIX.1 implementations are hosted in the sense of the C Standard {2}. See also the remarks on conformance in the Introduction.
- (6) *Text vs. binary file modes.* The C Standard {2} defines *text* and *binary* modes for a file. But the POSIX.1 interface and historical implementa-tions related to it make no such distinction, and all functions defined by POSIX.1 treat files as if these modes were identical. (It should not be stated that POSIX.1 files are either *text* or *binary*.) The definitions in the C Standard {2} were written so that this interpretation is possible. In particular, *text* mode files are not required to end with a line separator, which also means that they are not required to include a line separator at all.

Furthermore, there is a basic difference in approach between the Rationale accompanying the C Standard {2} and this Rationale Annex. The C Standard {2} Rationale, a separate document, addresses almost all changes as differences from the Base Documents of the C Standard {2}, usually either Kernighan and Ritchie {B46} or the 1984 */usr/group Standard* {B59}. This Rationale cannot do that, since there are many more variants of (and Base Documents for) the operating system interface than for the C language. The most noticeable aspect of this

difference is that the C Standard {2} Rationale identifies “QUIET CHANGES” from the Base Documents. This Annex cannot include such markings, since a quiet change from one historical implementation may correspond exactly to another historical implementation, and may be very noticeable to an application written for yet another.

The following subclauses justify the inclusion or omission of various C language functions in POSIX.1 or the C Standard {2}.

B.1.1.1.1 Solely by POSIX.1

These return parameters from the operating system environment: *ctermid()*, *ttyname()*, and *isatty()*.

The *fileno()* and *fdopen()* functions map between C language stream pointers and POSIX.1 file descriptors.

B.1.1.1.2 Solely by the C Standard

There are many functions that are useful with the operating system interface and are required for conformance with POSIX.1, but that are properly part of the C Language. These are listed in 8.1, which also notes which functions are defined by both POSIX.1 and the C Standard {2}. Certain terms defined by the C Standard {2} are incorporated by POSIX.1 in 2.7.

Some routines were considered too specialized to be included in POSIX.1. These include *bsearch()* and *qsort()*.

B.1.1.1.3 By Neither POSIX.1 Nor the C Standard

Some functions were considered of marginal utility and problematical when international character sets were considered: *_toupper()*, *_tolower()*, *toascii()*, and *isascii()*.

Although *malloc()* and *free()* are in the C Standard {2} and are required by 8.1 of POSIX.1, neither *brk()* nor *sbrk()* occur in either standard (although they were in the 1984 */usr/group Standard* {B59}), because POSIX.1 is designed to provide the basic set of functions required to write a Conforming POSIX.1 Application; the underlying implementation of *malloc()* or *free()* is not an appropriate concern for POSIX.1.

B.1.1.1.4 Base by POSIX.1, Additions by the C Standard

Since the C Standard {2} does not depend on POSIX.1 in any way, there are no items in this category.

B.1.1.1.5 Base by the C Standard, Additions by POSIX.1

The C Standard {2} has to define *errno* if only because examining that variable offers the only way to determine when some mathematics routines fail. But POSIX.1 uses it more extensively and adds some semantics to it in 2.4, which also defines some values for it.

Many numerical limits used by the C Standard {2} were incorporated by POSIX.1 in 2.8, and some new ones were added, all to be found in the header `<limits.h>`.

159 The C Standard {2} provides *signal()*, a minimal functionality for interrupts. The
 160 POSIX.1 definition replaces this with an elaborate mechanism that deals with
 161 multiple processes and is reliable when signals come from outside sources.

162 The *time()* function is used by the C Standard {2}, but POSIX.1 further specifies
 163 the time value.

164 The *getenv()* function is referenced in 2.6 and 3.1.2 and is also defined by the
 165 C Standard {2}.

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

168 The *setlocale()* function and the handling of time zones were further specified to
 169 take advantage of the POSIX environment.

170 The standard-I/O functions were specified in terms of their relationship to file
 171 descriptors and the relationship between multiple processes.

172 **B.1.1.1.6 Related Functions by Both**

173 The C Standard {2} definition of *compliance* and the POSIX.1 definition of *confor-*
 174 *mance* are similar, although the latter notes certain potential hardware
 175 limitations.

176 POSIX.1 defined a portable filename character set in 2.2.2 that is like the
 177 C Standard {2} identifier character set. However, POSIX.1 did not allow upper-
 178 and lowercase characters to be considered equivalent. See *filename portability* in
 179 2.3.4.

180 The *exit()* function is defined only by the C Standard {2} because it refers to clos-
 181 ing streams, and that subject, as well as *fclose()* itself, is defined almost entirely
 182 by the C Standard {2}. But POSIX.1 defined *_exit()*, which also adds semantics to
 183 *exit()*. This allows POSIX.1 to omit references to the C Standard {2} *atexit()*
 184 function.

185 POSIX.1 defined *kill()*, while the C Standard {2} defined *raise()*, which is similar
 186 except that it does not have a process ID argument, since the language defined by
 187 the C Standard {2} does not incorporate the idea of multiple processes.

188 The new functions *sigsetjmp()* and *siglongjmp()* were added to provide similar
 189 functions to the C Standard {2} *setjmp()* and *longjmp()* that additionally save and
 190 restore signal state.

191 **B.1.2 Normative References**

192 There is no additional rationale provided for this subclause.

B.1.3 Conformance

These conformance definitions are descended from those of *conforming implementation*, *conforming application*, and *conforming portable application* of early drafts, but were changed to clarify

- (1) Extensions, options, and limits;
- (2) Relations among the three terms, and;
- (3) Relations between POSIX.1 and the C Standard (2).

B.1.3.1 Implementation Conformance

These definitions allow application developers to know what to depend on in an implementation.

There is no definition of a *strictly conforming implementation*; that would be an implementation that provides *only* those facilities specified by POSIX.1 with no extensions whatsoever. This is because no actual operating system implementation can exist without system administration and initialization facilities that are beyond the scope of POSIX.1.

B.1.3.1.1 Requirements

The word “support” is used, rather than “provide,” in order to allow an implementation that has no resident software development facilities, but that supports the execution of a *Strictly Conforming POSIX.1 Application*, to be a *conforming implementation*. See also B.1.1.1.

B.1.3.1.2 Documentation

The conforming documentation is required to use the same numbering scheme as POSIX.1 for purposes of cross referencing. This requirement is consistent with and supplements the verification test assertions being developed by other POSIX groups. All options that an implementation chooses shall be reflected in `<limits.h>` and `<unistd.h>`.

Note that the use of “may” in terms of where conformance documents record where implementations may vary implies that it is not required to describe those features identified as undefined or unspecified.

Other aspects of systems must be evaluated by purchasers for suitability. Many systems incorporate buffering facilities, maintaining updated data in volatile storage and transferring such updates to nonvolatile storage asynchronously. Various exception conditions, such as a power failure or a system crash, can cause this data to be lost. The data may be associated with a file that is still open, with one that has been closed, with a directory, or with any other internal system data structures associated with permanent storage. This data can be lost, in whole or part, so that only careful inspection of file contents could determine that an update did not occur.

Also, interrelated file activities, where multiple files and/or directories are updated, or where space is allocated or released in the file system structures, can leave inconsistencies in the relationship between data in the various files and

234 directories, or in the file system itself. Such inconsistencies can break applica-
235 tions that expect updates to occur in a specific sequence, so that updates in one
236 place correspond with related updates in another place.

237 For example, if a user creates a file, places information in the file, and then
238 records this action in another file, a system or power failure at this point followed
239 by restart may result in a state in which the record of the action is permanently
240 recorded, but the file created (or some of its information) has been lost. The
241 consequences of this to the user may be undesirable. For a user on such a system,
242 the only safe action may be to require the system administrator to have a policy
243 that requires, after any system or power failure, that the entire file system must
244 be restored from the most recent backup copy (causing all intervening work to be
245 lost).

246 The characteristics of each implementation will vary in this respect and may or
247 may not meet the requirements of a given application or user. Enforcement of
248 such requirements is beyond the scope of POSIX.1. It is up to the purchaser to
249 determine what facilities are provided in an implementation that affect the expo-
250 sure to possible data or sequence loss and also what underlying implementation
251 techniques and/or facilities are provided that reduce or limit such loss or its
252 consequences.

253 **B.1.3.1.3 Conforming Implementation Options**

254 Within POSIX.1 there are some symbolic constants that, if defined, indicate that a
255 certain option is enabled. Other symbolic constants exist in POSIX.1 for other rea-
256 sons. This clause helps clarify which constants are related to true “options” and
257 which are related more to the behavior of differing systems.

258 To accommodate historical implementations where there were distinct semantics
259 in certain situations, but where one was not clearly better or worse than another,
260 early drafts of POSIX.1 permitted either of (typically) two options using “may.” At
261 the request of the working group developing test assertions, this was changed to
262 be specified by formal options with flags. It quickly became obvious that these
263 would be treated as options that could be selected by a purchaser, when the intent
264 of the developers of POSIX.1 was to allow either behavior (or both, in some cases)
265 to conform to the standard, and to constrain the application to accommodate
266 either. Thus, these options were removed and the phrase “An implementation
267 may either” introduced to replace the option. Where this phrase is used, it indi-
268 cates that an application shall tolerate either behavior.

269 It is intended that all conforming applications shall tolerate either behavior and
270 that only in the most exceptional of circumstances (driven by technical need)
271 should a purchaser specify only one behavior. Backwards compatibility is not con-
272 sidered exceptional, as this is not consistent with the intent of POSIX.1: to pro-
273 mote the portability of applications (and the development of portable
274 applications).

275 An application can tolerate these behaviors either by ignoring the differences (if
276 they are irrelevant to the application) or by taking an action to assure a known
277 state. It might be that that action would be redundant on some implementations.

278 Validation programs, which are applications in this sense, could either report the
279 actual result found or simply ignore the difference. In no case should either

acceptable behavior be treated as an error. This may complicate the validation slightly, but is more consistent with the intent of this permissible variation in behavior.

In certain circumstances, the behavior may vary for a given process. For example, in the presence of networked file systems, whether or not dot and dot-dot are present in the directory may vary with the directory being searched, and the program would only be portable if it tolerated, but did not require, the presence of these entries in a directory.

In situations like this, it is typically easier to simply ignore dot and dot-dot if they are found than to try to determine if they should be expected or not.

B.1.3.2 Application Conformance

These definitions guide users or adaptors of applications in determining on which implementations an application will run and how much adaptation would be required to make it run on others. These three definitions are modeled after related ones in the C Standard {2}.

POSIX.1 occasionally uses the expressions *portable application* or *conforming application*. As they are used, these are synonyms for any of these three terms. The differences between the three classes of application conformance relate to the requirements for other standards, or, in the case of the Conforming POSIX.1 Application Using Extensions, to implementation extensions. When one of the less explicit expressions is used, it should be apparent from the context of the discussion which of the more explicit names is appropriate.

B.1.3.2.1 Strictly Conforming POSIX.1 Application

This definition is analogous to that of a C Standard {2} *conforming program*.

The major difference between a *Strictly Conforming POSIX.1 Application* and a C Standard {2} *strictly conforming program* is that the latter is not allowed to use features of POSIX.1 that are not in the C Standard {2}.

B.1.3.2.2 Conforming POSIX.1 Application

Examples of <National Bodies> include ANSI, BSI, and AFNOR.

B.1.3.2.3 Conforming POSIX.1 Application Using Extensions

Due to possible requirements for configuration or implementation characteristics in excess of the specifications in 2.8 or related to the hardware (such as array size or file space), not every Conforming POSIX.1 Application Using Extensions will run on every conforming implementation.

B.1.3.3 Language-Dependent Services for the C Programming Language

POSIX.1 is, for historical reasons, both a specification of an operating system interface and a C binding for that specification. It is clear that these need to be separated into unique entities, but the urgency of getting the initial standard out, and the fact that C is the *de facto* primary language on systems similar to the

319 UNIX system, makes this a necessary and workable situation.

320 Nevertheless, work will be done on language bindings, beyond that for C before
321 the specification and the current binding are separated. Language bindings for
322 languages other than C should not model themselves too closely on the C binding
323 and in the process pick up various idiosyncrasies of C.

324 Where functionality is duplicated in POSIX.1 [e.g., *open()* and *creat()*] there is no
325 reason for that duplication to be carried forward into another language. On the
326 other hand, some languages have functionality already in them that is essentially
327 the same as that provided in POSIX.1. In this case, a mapping between the func-
328 tionality in that language and the underlying functionality in POSIX.1 is a better
329 choice than mimicking the C binding.

330 Since C has no syntax for I/O, and I/O is a large fraction of POSIX.1, the paradigm
331 of functions has been used. This may not be appropriate to another language.
332 For example, FORTRAN's REWIND statement is a candidate to map onto a special
333 case of *lseek()*, and its SEEK statement may completely cover for *lseek()*. If this is
334 the case, there is no reason to provide SUBROUTINES with the same functionality.
335 In the more general case, file descriptors and FORTRAN's logical unit numbers
336 may have a useful mapping. FORTRAN's ERR= option in I/O operations might
337 replace returning -1; the whole concept of errors might be handled differently.

338 As was done with C, it is not unreasonable for other language bindings to specify
339 some areas that are undefined or unspecified by the underlying language stan-
340 dard or that are permissible as extensions. This may, in fact, solve some difficult
341 problems.

342 Using as much as possible of the target language in the binding enhances porta-
343 bility. If a program wishes to use some POSIX.1 capabilities, and these are bound
344 to the language statements rather than appearing as additional procedure or
345 function calls, and the program does in fact conform to the language standard
346 while using those functions, it will port to a larger range of systems than one that
347 is obligated to use procedure or function calls introduced specifically for the bind-
348 ing to POSIX.1 to do the same thing.

349 A program that requires the POSIX.1 capabilities that are not bound to the stan-
350 dard language directly (as above) has no chance to be portable outside the POSIX.1
351 environment. It does not matter whether the extension is syntactic or a new func-
352 tion; it still will not port without effort. Given this, it seems unreasonable not to
353 consider language extensions when determining how best to map the functionality
354 of POSIX.1 into a particular language binding. For example, a new statement
355 similar to READ, which loads the values from a call like *stat()*, might be the best
356 solution for reading the data lists returned as structures in C into a list of FOR-
357 TRAN variables.

358 No attempt to mimic *printf()* or *scanf()* (or the rest of the C Standard {2} func-
359 tions) should be made; the equivalent functions in the language should be used.
360 (Formatted READ and WRITE in FORTRAN, *read/readln* and *write/writeln*
361 in Pascal, for example.)

362 There is an inherent special relationship between an operating system standard
363 and a language standard. It is unlikely that standards for other kinds of features
364 (such as graphics) will bind directly to statements in a general purpose language.

5 However, an operating system standard should provide the services required by a
6 language. This is an unusual situation, and the tendency to use only new func-
7 tions and procedures when creating a binding should be examined carefully. (A
8 one-to-one binding in all cases is probably not possible, but bindings such as those
9 for standard I/O in Section 8 may be possible.)

0 Binding directly to the language, where possible, should be encouraged both by
1 making maximal use of the mapping between the operating system and the
2 language that naturally exists and, where appropriate, by having the languages
3 request changes to the operating system to facilitate such a mapping. (A future
4 inclusion of a truncate function, specifically for the FORTRAN ENDFILE state-
5 ment, but that is also generally useful, is a good example.)

6 Part of the job of creating a binding is choosing names for functions that are intro-
7 duced, and these will need to be appropriate for that language. It is possible to
8 use other than the most restrictive form of a name, since, as discussed previously,
9 using these functions inherently makes the application not portable to systems
0 that are not POSIX.1, and if POSIX.1 conformant systems typically accept names
1 that the lowest-common-denominator system will not, there is no reason to *a*
2 *priori* exclude such names. (The specific example is C, where it is typically “non-
3 UNIX” systems that limit external identifiers to six characters.)

4 See B.1.1 for additional information about C bindings.

5 **B.1.3.3.1 Types of Conformance**

6 There is no additional rationale provided for this subclause.

7 **B.1.3.3.2 C Standard Language-Dependent System Support**

8 The issue of “namespace pollution” needs to be understood in this context. See
9 B.2.7.2.

10 **B.1.3.3.3 Common-Usage C Language-Dependent System Support**

11 The issue of “namespace pollution” needs to be understood in this context. See
12 B.2.7.2.

13 **B.1.3.4 Other C Language-Related Specifications**

14 The information concerning the use of library functions was adapted from a
15 description in the C Standard [2]. Here is an example of how an application pro-
16 gram can protect itself from library functions that may or may not be macros,
17 rather than true functions:

18 The *atoi()* function may be used in any of several ways:

- 19 (1) By use of its associated header (possibly generating a macro expansion)

```
20 #include <stdlib.h>  
21 /* ... */  
22 i = atoi(str);
```

- 23 (2) By use of its associated header (assuredly generating a true function call)

```

404         #include <stdlib.h>
405         #undef atoi
406         /* ... */
407         i = atoi(str);
408
409         or
410
409         #include <stdlib.h>
410         /* ... */
411         i = (atoi) (str);

```

412 (3) By explicit declaration

```

413         extern int atoi (const char *);
414         /* ... */
415         i = atoi(str);

```

416 (4) By implicit declaration

```

417         /* ... */
418         i = atoi(str);

```

419 (Assuming no function prototype is in scope. This is not allowed by the
 420 C Standard {2} for functions with variable arguments; furthermore,
 421 parameter type conversion “widening” is subject to different rules in this
 422 case.)

423 Note that the C Standard {2} reserves names starting with ‘_’ for the compiler.
 424 Therefore, the compiler could, for example, implement an intrinsic, built-in func-
 425 tion *_asm_builtin_atoi()*, which it recognized and expanded into inline assembly
 426 code. Then, in *<stdlib.h>*, there could be the following:

```

427         #define atoi(X) _asm_builtin_atoi(X)

```

428 The user’s “normal” call to *atoi()* would then be expanded inline, but the imple-
 429 mentor would also be required to provide a callable function named *atoi()* for use
 430 when the application requires it; for example, if its address is to be stored in a
 431 function pointer variable.

432 B.1.3.5 Other Language-Related Specifications

433 It is intended that “long” identifiers and multicase linkage would be supported on
 434 POSIX.1 systems for all languages, including C. This is where that condition is
 435 stated. The portion of the sentence about “if such extensions are” is included to
 436 permit languages that have an absolute maximum, or an absolute requirement of
 437 case folding, to be conformant.

438 The requirement for longer names is included for several reasons:

- 439 (1) Most systems similar to POSIX.1 are already conformant.
- 440 (2) Many existing language standards restrict the length of names to accom-
 441 modate existing systems that cannot be modified to allow longer names.
 442 However, those systems are not expected to be POSIX.1 conformant, for
 443 other reasons.

14 (3) Many historical applications rely on such long names. |

15 (4) Future languages (such as FORTRAN 88) are likely to require it. |

16 Specific to FORTRAN 77 {B21}, that standard permits long names, and this part of |
17 ISO/IEC 9945 requires that FORTRAN implementations running on POSIX.1 sup- |
18 port long names. The requirements of case distinction and length are considered |
19 orthogonal, but both are required if both are permitted by the language. Note |
20 that a language can be conformant to POSIX.1 even though a binding does not |
21 exist, because an application need not step outside the language standard to write |
22 a useful program. |

23 This requirement permits the use of reasonable-length names in a POSIX.1 bind- |
24 ing to a language such as FORTRAN. Clearly nothing prohibits a program that |
25 does conform to the FORTRAN minima to compile and run on POSIX.1. |

26 It is within the constraints of POSIX.1 to specify the behavior of the language pro- |
27 cessors and linker, consistent with the language, as it is a specification for an exe- |
28 cution environment. This is different than a package such as GKS {B27}, which |
29 can reasonably be expected to be ported to a system that enforces the language |
30 minima. |

31 It might be argued that this specification is appropriate to the language binding |
32 committees for POSIX generally, rather than specifically to POSIX.1. That argu- |
33 ment misses the intent. The intent is to require that the linker and other code |
34 that handles “object code” (a concept not formally defined in POSIX.1) are able to |
35 support long names. This requirement, being one that spans all languages, |
36 belongs in the specification standard, rather than tied to any one language. Note |
37 that it is also somewhat permissive, in that if the language is unable to deal with |
38 long names it is permitted not to require them, but it does remove the argument |
39 that “the loader might not permit long names, so [a specific] language binding |
40 should not force the issue.” |

41 A strictly conforming application for a given language could not use any exten- |
42 sions outside of POSIX.1 for that language (regardless of the underlying operating |
43 system). An application will strictly conform to POSIX.1 if it conforms to the |
44 language using additional interfaces from that language’s binding to POSIX.1. |

75 **B.2 Definitions and General Requirements**

76 **B.2.1 Conventions**

77 There is no additional rationale provided for this subclause.

478 **B.2.2 Definitions**479 **B.2.2.1 Terminology**

480 The meanings specified in POSIX.1 for the words *shall*, *should*, and *may* are man-
481 dated by ISO/IEC directives.

482 In this Rationale, the words *shall*, *should*, and *may* are sometimes used to illus-
483 trate similar usages in the standard. However, the Rationale itself does not
484 specify anything regarding implementations or applications.

485 **conformance document:** As a practical matter, the conformance document is
486 effectively part of the system documentation. They are distinguished by POSIX.1
487 so that they can be referred to distinctly.

488 **implementation defined:** This definition is analogous to that of the
489 C Standard [2] and, together with *undefined* and *unspecified*, provides a range of
490 specification of freedom allowed to the interface implementor.

491 **may:** The use of *may* has been limited as much as possible, due both to confu-
492 sion stemming from its ordinary English meaning and to objections regarding the
493 desirability of having as few options as possible and those as clearly specified as
494 possible.

495 **shall:** Declarative sentences are sometimes used in POSIX.1 as if they included
496 the word *shall*, and facilities thus specified are no less required. For example, the
497 two statements:

498 (1) The *foo()* function shall return zero

499 (2) The *foo()* function returns zero

500 are meant to be exactly equivalent. It is expected that a future version of POSIX.1
501 will be rewritten to use the “shall” form more consistently.

502 **should:** In POSIX.1, the word *should* does not usually apply to the implementa-
503 tion, but rather to the application. Thus, the important words regarding imple-
504 mentations are *shall*, which indicates requirements, and *may*, which indicates
505 options.

506 **obsolescent:** The term *obsolescent* was preferred over *deprecated* to represent
507 functionality that should not be used in new work. The term *obsolescent* is more
508 intuitive and reduced the possibility of misunderstanding in the intended context.

509 **supported:** An example of this concept is the *setpgid()* function. If the imple-
510 mentation does not support the optional job control feature, it nevertheless has to
511 provide a function named *setpgid()*, even though its only ability is that of return-
512 ing [ENOSYS].

513 **system documentation:** The system documentation should normally describe
514 the whole of the implementation, including any extensions provided by the imple-
515 mentation. Such documents normally contain information at least as detailed as
516 the POSIX.1 specifications. Few requirements are made on the system documen-
517 tation, but the term is needed to avoid a dangling pointer where the conformance
518 document is permitted to point to the system documentation.

undefined: See *implementation defined*.

unspecified: See *implementation defined*.

The definitions for *unspecified* and *undefined* appear nearly identical at first examination, but are not. *Unspecified* means that a conforming program may deal with the unspecified behavior, and it should not care what the outcome is. *Undefined* says that a conforming program should not do it because no definition is provided for what it does (and implicitly it would care what the outcome was if it tried it). It is important to remember, however, that if the syntax permits the statement at all, it must have some outcome in a real implementation.

Thus, the terms *undefined* and *unspecified* apply to the way the application should think about the feature. In terms of the implementation it is always “defined”—there is always some result, even if it is an error. The implementation is free to choose the behavior it prefers.

This also implies that an implementation, or another standard, could specify or define the result in a useful fashion. The terms apply to POSIX.1 specifically.

The term *implementation defined* implies requirements for documentation that are not required for *undefined* (or *unspecified*). Where there is no need for a conforming program to know the definition, the term *undefined* is used, even though *implementation defined* could also have been used in this context. There could be a fourth term, specifying “POSIX.1 does not say what this does; it is acceptable to define it in an implementation, but it does not need to be documented,” and *undefined* would then be used very rarely for the few things for which any definition is not useful.

In many places POSIX.1 is silent about the behavior of some possible construct. For example, a variable may be defined for a specified range of values and behaviors are described for those values; nothing is said about what happens if the variable has any other value. That kind of silence can imply an error in the standard, but it may also imply that the standard was intentionally silent and that any behavior is permitted. There is a natural tendency to infer that if the standard is silent, a behavior is prohibited. That is not the intent. Silence is intended to be equivalent to the term *unspecified*.

B.2.2.2 General Terms

Many of these definitions are necessarily circular, and some of the terms (such as *process*) are variants of basic computing science terms that are inherently hard to define. Some are defined by context in the prose topic descriptions of the general concepts in 2.3, but most appear in the alphabetical glossary format of the terms in 2.2.2.

Some definitions must allow extension to cover terms or facilities that are not explicitly mentioned in POSIX.1. For example, the definition of *file* must permit interpretation to include streams, as found in the Eighth Edition (a research version of the UNIX system). The use of abstract intermediate terms (such as *object* in place of, or in addition to, *file*) has mostly been avoided in favor of careful definition of more traditional terms.

562 Some terms in the following list of notes do not appear in POSIX.1; these are
563 marked prefixed with an asterisk (*). Many of them have been specifically
564 excluded from POSIX.1 because they concern system administration, implementa-
565 tion, or other issues that are not specific to the programming interface. Those are
566 marked with a reason, such as “implementation defined.”

567 **appropriate privileges:** One of the fundamental security problems with many
568 historical UNIX systems has been that the privilege mechanism is monolithic—a
569 user has either no privileges or *all* privileges. Thus, a successful “trojan horse”
570 attack on a privileged process defeats all security provisions. Therefore, POSIX.1
571 allows more granular privilege mechanisms to be defined. For many historical
572 implementations of the UNIX system, the presence of the term *appropriate*
573 *privileges* in POSIX.1 may be understood as a synonym for *super-user* (UID 0).
574 However, future systems will undoubtedly emerge where this is not the case and
575 each discrete controllable action will have *appropriate privileges* associated with
576 it. Because this mechanism is *implementation defined*, it must be described in
577 the conformance document. Although that description affects several parts of
578 POSIX.1 where the term *appropriate privilege* is used, because the term *implemen-*
579 *tation defined* only appears here, the description of the entire mechanism and its
580 effects on these other sections belongs in clause 2.3 of the conformance document.
581 This is especially convenient for implementations with a single mechanism that
582 applies in all areas, since it only needs to be described once.

583 **clock tick:** The C Standard [2] defines a similar interval for use by the *clock()*
584 function. There is no requirement that these intervals be the same. In historical
585 implementations these intervals are different. Currently only the *times()* function
586 uses values stated in terms of clock ticks, although other functions might use
587 them in the future.

588 **controlling terminal:** The question of which of possibly several special files
589 referring to the terminal is meant is not addressed in POSIX.1.

590

591 ***device number:** The concept is handled in *stat()* as *ID of device*.

592 **directory:** The format of the directory file is implementation defined and differs
593 radically between System V and 4.3BSD. However, routines (derived from
594 4.3BSD) for accessing directories are provided in 5.1.2 and certain constraints on
595 the format of the information returned by those routines are made in 5.1.1.

596 **directory entry:** Throughout the document, the term *link* is used [about the
597 *link()* function, for example] in describing the objects that point to files from
598 directories.

599 **dot:** The symbolic name *dot* is carefully used in POSIX.1 to distinguish the work-
600 ing directory filename from a period or a decimal point.

601 **dot-dot:** Historical implementations permit the use of these filenames without
602 their special meanings. Such use precludes any meaningful use of these
603 filenames by a Conforming POSIX.1 Application. Therefore, such use is considered
604 an extension, the use of which makes an implementation nonconforming. See also
605 B.2.3.7.

Epoch: Historically, the origin of UNIX system time was referred to as “00:00:00 GMT, January 1, 1970.” Greenwich Mean Time is actually not a term acknowledged by the international standards community; therefore, this term, *Epoch*, is used to abbreviate the reference to the actual standard, Coordinated Universal Time. The concept of leap seconds is added for precision; at the time POSIX.1 was published, 14 leap seconds had been added since January 1, 1970. These 14 seconds are ignored to provide an easy and compatible method of computing time differences.

Most systems’ notion of “time” is that of a continuously increasing value, so this value should increase even during leap seconds. However, not only do most systems not keep track of leap seconds, but most systems are probably not synchronized to any standard time reference. Therefore, it is inappropriate to require that a time represented as seconds since the Epoch precisely represent the number of seconds between the referenced time and the Epoch.

It is sufficient to require that applications be allowed to treat this time as if it represented the number of seconds between the referenced time and the Epoch. It is the responsibility of the vendor of the system, and the administrator of the system, to ensure that this value represents the number of seconds between the referenced time and the Epoch as closely as necessary for the application being run on that system.

It is important that the interpretation of time names and *seconds since the Epoch* values be consistent across conforming systems. That is, it is important that all conforming systems interpret “536457599 seconds since the Epoch” as 59 seconds, 59 minutes, 23 hours 31 December 1986, regardless of the accuracy of the system’s idea of the current time. The expression is given to assure a consistent interpretation, not to attempt to specify the calendar. The relationship between *tm_yday* and the day of week, day of month, and month is presumed to be specified elsewhere and is not given in POSIX.1.

Consistent interpretation of *seconds since the Epoch* can be critical to certain types of distributed applications that rely on such timestamps to synchronize events. The accrual of leap seconds in a time standard is not predictable. The number of leap seconds since the Epoch will likely increase. POSIX.1 is more concerned about the synchronization of time between applications of astronomically short duration. These concerns are expected to become more critical in the future.

Note that *tm_yday* is zero-based, not one-based, so the day number in the example above is 364. Note also that the division is an integer division (discarding remainder) as in the C language.

Note also that in Section 8, the meaning of *gmtime()*, *localtime()*, and *mktime()* is specified in terms of this expression. However, the C Standard [2] computes *tm_yday* from *tm_mday*, *tm_mon*, and *tm_year* in *mktime()*. Because it is stated as a (bidirectional) relationship, not a function, and because the conversion between month-day-year and day-of-year dates is presumed well known and is also a relationship, this is not a problem.

Note that the expression given will fail after the year 2099. Since the issue of *time_t* overflowing a 32-bit integer occurs well before that time, both of these will have to be addressed in revisions to POSIX.1.

652 **FIFO special file:** See *pipe* in B.2.2.2.

653 **file:** It is permissible for an implementation-defined file type to be nonreadable
654 or nonwritable.

655 **file classes:** These classes correspond to the historical sets of permission bits.
656 The classes are general to allow implementations flexibility in expanding the
657 access mechanism for more stringent security environments. Note that a process
658 is in one and only one class, so there is no ambiguity.

659 **filename:** At the present time, the primary responsibility for truncating
660 filenames containing multibyte characters must reside with the application. |
661 Some industry groups involved in internationalization believe that in the future |
662 the responsibility must reside with the kernel. For the moment, a clearer under-
663 standing of the implications of making the kernel responsible for truncation of
664 multibyte file names is needed.

665 Character level truncation was not adopted because there is no support in |
666 POSIX.1 that advises how the kernel distinguishes between single and multibyte |
667 characters. Until that time, it must be incumbent upon application writers to
668 determine where multibyte characters must be truncated.

669 **file system:** Historically the meaning of this term has been overloaded with two
670 meanings: that of the complete file hierarchy and that of a mountable subset of
671 that hierarchy; i.e., a mounted file system. POSIX.1 uses the term *file system* in
672 the second sense, except that it is limited to the scope of a process (and a process's
673 root directory). This usage also clarifies the domain in which a file serial number
674 is unique.

675 ***group file:** Implementation defined; see B.9.

676 ***historical implementations:** This refers to previously existing implementa-
677 tions of programming interfaces and operating systems that are related to the
678 interface specified by POSIX.1. See also "Minimal Changes to Historical Imple- |
679 mentations" in the Introduction. |

680 ***hosted implementation:** This refers to a POSIX.1 implementation that is
681 accomplished through interfaces from the POSIX.1 services to some alternate form
682 of operating system kernel services. Note that the line between a hosted imple-
683 mentation and a native implementation is blurred, since most implementations
684 will provide some services directly from the kernel and others through some
685 indirect path. [For example, *fopen()* might use *open()*; or *mkfifo()* might use
686 *mknod()*.] There is no necessary relationship between the type of implementation
687 and its correctness, performance, and/or reliability.

688 ***implementation:** The term is generally used instead of its synonym, *system*,
689 to emphasize the consequences of decisions to be made by system implementors.
690 Perhaps if no options or extensions to POSIX.1 were allowed, this usage would not
691 have occurred.

692 The term *specific implementation* is sometimes used as a synonym for *implemen-*
693 *tation*. This should not be interpreted too narrowly; both terms can represent a
694 relatively broad group of systems. For example, a hardware vendor could market
695 a very wide selection of systems that all used the same instruction set, with some
696 systems desktop models and others large multiuser minicomputers. This wide

range would probably share a common POSIX.1 operating system, allowing an application compiled for one to be used on any of the others; this is a *[specific] implementation*.

However, that wide range of machines probably has some differences between the models. Some may have different clock rates, different file systems, different resource limits, different network connections, etc., depending on their sizes or intended usages. Even on two identical machines, the system administrators may configure them differently. Each of these different systems is known by the term *a specific instance of a specific implementation*. This term is only used in the portions of POSIX.1 dealing with run-time queries: *sysconf()* and *pathconf()*.

***incomplete pathname:** Absolute pathname has been adequately defined.

job control: In order to understand the job-control facilities in POSIX.1 it is useful to understand how they are used by a job-control-cognizant shell to create the user interface effect of job control.

While the job-control facilities supplied by POSIX.1 can, in theory, support different types of interactive job-control interfaces supplied by different types of shells, there is historically one particular interface that is most common (provided by BSD C Shell). This discussion describes that interface as a means of illustrating how the POSIX.1 job-control facilities can be used.

Job control allows users to selectively stop (suspend) the execution of processes and continue (resume) their execution at a later point. The user typically employs this facility via the interactive interface jointly supplied by the terminal I/O driver and a command interpreter (shell).

The user can launch jobs (command pipelines) in either the foreground or background. When launched in the foreground, the shell waits for the job to complete before prompting for additional commands. When launched in the background, the shell does not wait, but immediately prompts for new commands.

If the user launches a job in the foreground and subsequently regrets this, the user can type the suspend character (typically set to control-Z), which causes the foreground job to stop and the shell to begin prompting for new commands. The stopped job can be continued by the user (via special shell commands) either as a foreground job or as a background job. Background jobs can also be moved into the foreground via shell commands.

If a background job attempts to access the login terminal (controlling terminal), it is stopped by the terminal driver and the shell is notified, which, in turn, notifies the user. [Terminal access includes *read()* and certain terminal control functions and conditionally includes *write()*.] The user can continue the stopped job in the foreground, thus allowing the terminal access to succeed in an orderly fashion. After the terminal access succeeds, the user can optionally move the job into the background via the suspend character and shell commands.

Implementing Job Control Shells

The interactive interface described previously can be accomplished using the POSIX.1 job-control facilities in the following way.

The key feature necessary to provide job control is a way to group processes into jobs. This grouping is necessary in order to direct signals to a single job and also

742 to identify which job is in the foreground. (There is at most one job that is in the
743 foreground on any controlling terminal at a time.)

744 The concept of *process groups* is used to provide this grouping. The shell places
745 each job in a separate process group via the *setpgid()* function. To do this, the
746 *setpgid()* function is invoked by the shell for each process in the job. It is actually
747 useful to invoke *setpgid()* twice for each process: once in the child process, after
748 calling *fork()* to create the process, but before calling one of the *exec* functions to
749 begin execution of the program, and once in the parent shell process, after calling
750 *fork()* to create the child. The redundant invocation avoids a race condition by
751 ensuring that the child process is placed into the new process group before either
752 the parent or the child relies on this being the case. The *process group ID* for the
753 job is selected by the shell to be equal to the *process ID* of one of the processes in
754 the job. Some shells choose to make one process in the job be the parent of the
755 other processes in the job (if any). Other shells (e.g., the C Shell) choose to make
756 themselves the parent of all processes in the pipeline (job). In order to support
757 this latter case, the *setpgid()* function accepts a process group ID parameter since
758 the correct process group ID cannot be inherited from the shell. The shell itself is
759 considered to be a job and is the sole process in its own process group.

760 The shell also controls which job is currently in the foreground. A foreground and
761 background job differ in two ways: the shell waits for a foreground command to
762 complete (or stop) before continuing to read new commands, and the terminal I/O
763 driver inhibits terminal access by background jobs (causing the processes to stop).
764 Thus, the shell must work cooperatively with the terminal I/O driver and have a
765 common understanding of which job is currently in the foreground. It is the user
766 who decides which command should be currently in the foreground, and the user
767 informs the shell via shell commands. The shell, in turn, informs the terminal I/O
768 driver via the *tcsetpgrp()* function. This indicates to the terminal I/O driver the
769 process group ID of the foreground process group (job). When the current fore-
770 ground job either stops or terminates, the shell places itself in the foreground via
771 *tcsetpgrp()* before prompting for additional commands. Note that when a job is
772 created the new process group begins as a background process group. It requires
773 an explicit act of the shell via *tcsetpgrp()* to move a process group (job) into the
774 foreground.

775 When a process in a job stops or terminates, its parent (e.g., the shell) receives
776 synchronous notification by calling the *waitpid()* function with the WUNTRACED
777 flag set. Asynchronous notification is also provided when the parent establishes a
778 signal handler for SIGCHLD and does not specify the SA_NOCLDSTOP flag. Usu-
779 ally all processes in a job stop as a unit since the terminal I/O driver always sends
780 job-control stop signals to all processes in the process group.

781 To continue a stopped job, the shell sends the SIGCONT signal to the process
782 group of the job. In addition, if the job is being continued in the foreground, the
783 shell invokes *tcsetpgrp()* to place the job in the foreground before sending
784 SIGCONT. Otherwise, the shell leaves itself in the foreground and reads addi-
785 tional commands.

786 There is additional flexibility in the POSIX.1 job-control facilities that allows devi-
787 ations from the typical interface. Clearing the TOSTOP terminal flag (see 7.1.2.5)
788 allows background jobs to perform *write()* functions without stopping. The same
789 effect can be achieved on a per-process basis by having a process set the signal
790 action for SIGTTOU to SIG_IGN.

Note that the terms *job* and *process group* can be used interchangeably. A login session that is not using the job control facilities can be thought of as a large collection of processes that are all in the same job (process group). Such a login session may have a partial distinction between foreground and background processes; that is, the shell may choose to wait for some processes before continuing to read new commands and may not wait for other processes. However, the terminal I/O driver will consider all these processes to be in the foreground since they are all members of the same process group.

In addition to the basic job-control operations already mentioned, a job-control-cognizant shell needs to perform the following actions:

When a foreground (not background) job stops, the shell must sample and remember the current terminal settings so that it can restore them later when it continues the stopped job in the foreground [via the *tcgetattr()* and *tcsetattr()* functions].

Because a shell itself can be spawned from a shell, it must take special action to ensure that subshells interact well with their parent shells.

A subshell can be spawned to perform an interactive function (prompting the terminal for commands) or a noninteractive function (reading commands from a file). When operating noninteractively, the job-control shell will refrain from performing the job-control specific actions described above. It will behave as a shell that does not support job control. For example, all *jobs* will be left in the same process group as the shell, which itself remains in the process group established for it by its parent. This allows the shell and its children to be treated as a single job by a parent shell, and they can be affected as a unit by terminal keyboard signals.

An interactive subshell can be spawned from another job-control-cognizant shell in either the foreground or background. (For example, from the C Shell, the user can execute the command, *csch &.*) Before the subshell activates job control by calling *setpgid()* to place itself in its own process group and *tcsetpgrp()* to place its new process group in the foreground, it needs to ensure that it has already been placed in the foreground by its parent. (Otherwise, there could be multiple job-control shells that simultaneously attempt to control mediation of the terminal.) To determine this, the shell retrieves its own process group via *getpgrp()* and the process group of the current foreground job via *tcgetpgrp()*. If these are not equal, the shell sends SIGTTIN to its own process group, causing itself to stop. When continued later by its parent, the shell repeats the process-group check. When the process groups finally match, the shell is in the foreground and it can proceed to take control. After this point, the shell ignores all the job-control stop signals so that it does not inadvertently stop itself.

Implementing Job Control Applications

Most applications do not need to be aware of job-control signals and operations; the intuitively correct behavior happens by default. However, sometimes an application can inadvertently interfere with normal job-control processing, or an application may choose to overtly effect job control in cooperation with normal shell procedures.

An application can inadvertently subvert job-control processing by “blindly” altering the handling of signals. A common application error is to learn how many

837 signals the system supports and to ignore or catch them all. Such an application
838 makes the assumption that it does not know what this signal is, but knows the
839 right handling action for it. The system may initialize the handling of job-control
840 stop signals so that they are being ignored. This allows shells that do not support
841 job control to inherit and propagate these settings and hence to be immune to stop
842 signals. A job-control shell will set the handling to the default action and pro-
843 propagate this, allowing processes to stop. In doing so, the job-control shell is taking
844 responsibility for restarting the stopped applications. If an application wishes to
845 catch the stop signals itself, it should first determine their inherited handling
846 states. If a stop signal is being ignored, the application should continue to ignore
847 it. This is directly analogous to the recommended handling of SIGINT described in
848 the UNIX Programmer's Manual [B41].

849 If an application is reading the terminal and has disabled the interpretation of
850 special characters (by clearing the ISIG flag), the terminal I/O driver will not send
851 SIGTSTP when the suspend character is typed. Such an application can simulate
852 the effect of the suspend character by recognizing it and sending SIGTSTP to its
853 process group as the terminal driver would have done. Note that the signal is
854 sent to the process group, not just to the application itself; this ensures that other
855 processes in the job also stop. (Note also that other processes in the job could be
856 children, siblings, or even ancestors.) Applications should not assume that the
857 suspend character is control-Z (or any particular value); they should retrieve the
858 current setting at startup.

859 *Implementing Job Control Systems*

860 The intent in adding 4.2BSD-style job control functionality was to adopt the neces-
861 sary 4.2BSD programmatic interface with only minimal changes to resolve syntac-
862 tic or semantic conflicts with System V or to close recognized security holes. The
863 goal was to maximize the ease of providing both conforming implementations and
864 Conforming POSIX.1 Applications.

865 Discussions of the changes can be found in the clauses that discuss the specific
866 interfaces. See B.3.2.1, B.3.2.2, B.3.3.1.1, B.3.3.2, B.3.3.4, B.4.3.1, B.4.3.3,
867 B.7.1.1.4, and B.7.2.4.

868 It is only useful for a process to be affected by job-control signals if it is the des-
869 cendant of a job-control shell. Otherwise, there will be nothing that continues the
870 stopped process. Because a job-control shell is allowed, but not required, by
871 POSIX.1, an implementation must provide a mechanism that shields processes
872 from job-control signals when there is no job-control shell. The usual method is
873 for the system initialization process (typically called `init`), which is the ancestor
874 of all processes, to launch its children with the signal handling action set to
875 SIG_IGN for the signals SIGTSTP, SIGTTIN, and SIGTTOU. Thus, all login shells
876 start with these signals ignored. If the shell is not job-control cognizant, then it
877 should not alter this setting and all its descendants should inherit the same
878 ignored settings. At the point where a job-control shell is launched, it resets the
879 signal handling action for these signals to be SIG_DFL for its children and (by
880 inheritance) their descendants. Also, shells that are not job-control cognizant will
881 not alter the process group of their descendants or of their controlling terminal;
882 this has the effect of making all processes be in the foreground (assuming the
883 shell is in the foreground). While this approach is valid, POSIX.1 added the con-
884 cept of orphaned process groups to provide a more robust solution to this problem.

All processes in a session managed by a shell that is not job-control cognizant are in an orphaned process group and are protected from stopping.

POSIX.1 does not specify how controlling terminal access is affected by a user logging out (that is, by a controlling process terminating). 4.2BSD uses the *vhangup()* function to prevent any access to the controlling terminal through file descriptors opened prior to logout. System V does not prevent controlling terminal access through file descriptors opened prior to logout (except for the case of the special file, */dev/tty*). Some implementations choose to make processes immune from job control after logout (that is, such processes are always treated as if in the foreground); other implementations continue to enforce foreground/background checks after logout. Therefore, a Conforming POSIX.1 Application should not attempt to access the controlling terminal after logout since such access is unreliable. If an implementation chooses to deny access to a controlling terminal after its controlling process exits, POSIX.1 requires a certain type of behavior (see 7.1.1.3).

***kernel:** See *system call*.

***library routine:** See *system call*.

***logical device:** Implementation defined.

***mount point:** The directory on which a *mounted file system* is mounted. This term, like *mount()* and *umount()*, was not included because it was implementation defined.

***mounted file system:** See *file system*.

***native implementation:** This refers to an implementation of POSIX.1 that interfaces directly to an operating-system kernel. See also *hosted implementation* and *cooperating implementation*. A similar concept is a native UNIX system, which would be a kernel derived from one of the original UNIX system products.

open file description: An *open file description*, as it is currently named, describes how a file is being accessed. What is currently called a *file descriptor* is actually just an identifier or “handle”; it does not actually describe anything.

The following alternate names were discussed:

For *open file description*:

open instance, *file access description*, *open file information*, and *file access information*.

For *file descriptor*:

file handle, *file number* [c.f., *fileno()*]. Some historical implementations use the term *file table entry*.

orphaned process group: Historical implementations have a concept of an orphaned process, which is a process whose parent process has exited. When job control is in use, it is necessary to prevent processes from being stopped in response to interactions with the terminal after they no longer are controlled by a job-control-cognizant program. Because signals generated by the terminal are sent to a process group and not to individual processes, and because a signal may be provoked by a process that is not orphaned, but sent to another process that is orphaned, it is necessary to define an orphaned process group. The definition

929 assumes that a process group will be manipulated as a group and that the job-
 930 control-cognizant process controlling the group is outside of the group and is the
 931 parent of at least one process in the group [so that state changes may be reported
 932 via *waitpid()*]. Therefore, a group is considered to be controlled as long as at least
 933 one process in the group has a parent that is outside of the process group, but
 934 within the session.

935 This definition of orphaned process groups ensures that a session leader's process
 936 group is always considered to be orphaned, and thus it is prevented from stopping
 937 in response to terminal signals.

938 ***passwd file:** Implementation defined; see B.9.

939 **parent directory:** There may be more than one directory entry pointing to a
 940 given directory in some implementations. The wording here identifies that
 941 exactly one of those is the parent directory. In 2.3.6, *dot-dot* is identified as the
 942 way that the unique directory is identified. (That is, the parent directory is the
 943 one to which dot-dot points.) In the case of a remote file system, if the same file
 944 system is mounted several times, it would appear as if they were distinct file sys-
 945 tems (with interesting synchronization properties).

946 **pipe:** It proved convenient to define a *pipe* as a special case of a *FIFO* even
 947 though historically the latter was not introduced until System III and does not
 948 exist at all in 4.3BSD.

949 **portable filename character set:** The encoding of this character set is not
 950 specified—specifically, ASCII is not required. But the implementation must pro-
 951 vide a unique character code for each of the printable graphics specified by
 952 POSIX.1. See also B.2.3.5.

953 Situations where characters beyond the portable filename character set (or histor-
 954 ically ASCII or ISO/IEC 646 {1}) would be used (in a context where the portable
 955 filename character set or ISO/IEC 646 {1} is required by POSIX.1) are expected to
 956 be common. Although such a situation renders the use technically noncompliant,
 957 mutual agreement among the users of an extended character set will make such
 958 use portable between those users. Such a mutual agreement could be formalized
 959 as an optional extension to POSIX.1. (Making it required would eliminate too
 960 many possible systems, as even those systems using ISO/IEC 646 {1} as a base
 961 character set extend their character sets for Western Europe and the rest of the
 962 world in different ways.)

963 Nothing in POSIX.1 is intended to preclude the use of extended characters where
 964 interchange is not required or where mutual agreement is obtained. It has been
 965 suggested that in several places “should” be used instead of “shall.” Because (in
 966 the worst case) use of any character beyond the portable filename character set
 967 would render the program or data not portable to all possible systems, no exten-
 968 sions are permitted in this context.

969 **regular file:** POSIX.1 does not intend to preclude the addition of structuring
 970 data (e.g., record lengths) in the file, as long as such data is not visible to an
 971 application that uses the features described in POSIX.1.

972 **root directory:** This definition permits the operation of *chroot()*, even though
 973 that function is not in POSIX.1. See also *file hierarchy*.

***root file system:** Implementation defined.

***root of a file system:** Implementation defined. See *mount point*.

seconds since the Epoch: The formula here is not precisely correct for leap centuries. See the discussion for *Epoch* for further details.

signal: The definition implies a double meaning for the term. Although a signal is an event, common usage implies that a signal is an identifier of the class of event.

***system call:** The distinction between a *system call* and a *library routine* is an implementation detail that may differ between implementations and has thus been excluded from POSIX.1. See “Interface, Not Implementation” in the Introduction.

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

B.2.2.3 Abbreviations

There is no additional rationale provided for this subclause.

B.2.3 General Concepts

B.2.3.1 extended security controls: Allowing an implementation to define extended security controls enables the use of POSIX.1 in environments that require different or more rigorous security than that provided in POSIX.1. Extensions are allowed in two areas: privilege and file access permissions. The semantics of these areas have been defined to permit extensions with reasonable, but not exact, compatibility with all existing practices. For example, the elimination of the super-user definition precludes identifying a process as privileged or not by virtue of its effective user ID.

B.2.3.2 file access permissions: A process should not try to anticipate the result of an attempt to access data by *a priori* use of these rules. Rather, it should make the attempt to access data and examine the return value (and possibly *errno* as well), or use *access()*. An implementation may include other security mechanisms in addition to those specified in POSIX.1, and an access attempt may fail because of those additional mechanisms, even though it would succeed according to the rules given in this subclause. (For example, the user’s security level might be lower than that of the object of the access attempt.) The optional supplementary group IDs provide another reason for a process to not attempt to anticipate the result of an access attempt.

B.2.3.3 file hierarchy: Though the file hierarchy is commonly regarded to be a tree, POSIX.1 does not define it as such for three reasons:

(1) Links may join branches.

(2) In some network implementations, there may be no single absolute root directory. See *pathname resolution*.

- 1013 (3) With symbolic links (found in 4.3BSD), the file system need not be a tree
1014 or even a directed acyclic graph.

1015 **B.2.3.4 file permissions:** Examples of implementation-defined constraints that
1016 may deny access are mandatory labels and access control lists.

1017 **B.2.3.5 filename portability:** Historically, certain filenames have been
1018 reserved. This list includes `core`, `/etc/passwd`, etc. Portable applications
1019 should avoid these.

1020 Most historical implementations prohibit case folding in filenames; i.e., treating
1021 upper- and lowercase alphabetic characters as identical. However, some consider
1022 case folding desirable:

- 1023 — For user convenience
- 1024 — For ease of implementation of the POSIX.1 interface as a hosted system on
1025 some popular operating systems, which is compatible with the goal of mak-
1026 ing the POSIX.1 interface broadly implementable (see “Broadly Implement-
1027 able” in the Introduction)

1028 Variants such as maintaining case distinctions in filenames, but ignoring them in
1029 comparisons, have been suggested. Methods of allowing escaped characters of the
1030 case opposite the default have been proposed

1031 Many reasons have been expressed for not allowing case folding, including:

- 1032 (1) No solid evidence has been produced as to whether case sensitivity or
1033 case insensitivity is more convenient for users.
- 1034 (2) Making case insensitivity a POSIX.1 implementation option would be
1035 worse than either having it or not having it, because
 - 1036 (a) More confusion would be caused among users.
 - 1037 (b) Application developers would have to account for both cases in their
1038 code.
 - 1039 (c) POSIX.1 implementors would still have other problems with native
1040 file systems, such as short or otherwise constrained filenames or
1041 pathnames, and the lack of hierarchical directory structure.
- 1042 (3) Case folding is not easily defined in many European languages, both
1043 because many of them use characters outside the USASCII alphabetic set,
1044 and because
 - 1045 (a) In Spanish, the digraph `ll` is considered to be a single letter, the
1046 capitalized form of which may be either `Ll` or `LL`, depending on con-
1047 text.
 - 1048 (b) In French, the capitalized form of a letter with an accent may or
1049 may not retain the accent depending on the country in which it is
1050 written.
 - 1051 (c) In German, the sharp ess may be represented as a single character
1052 resembling a Greek beta (β) in lowercase, but as the digraph `ss` in
1053 uppercase.

- 54 (d) In Greek, there are several lowercase forms of some letters; the one
55 to use depends on its position in the word. Arabic has similar rules.
- 56 (4) Many East Asian languages, including Japanese, Chinese, and Korean,
57 do not distinguish case and are sometimes encoded in character sets that
58 use more than one byte per character.
- 59 (5) Multiple character codes may be used on the same machine simultane-
60 ously. There are several ISO character sets for European alphabets. In
61 Japan, several Japanese character codes are commonly used together,
62 sometimes even in filenames; this is evidently also the case in China. To
63 handle case insensitivity, the kernel would have to at least be able to dis-
64 tinguish for which character sets the concept made sense.
- 65 (6) The file system implementation historically deals only with bytes, not
66 with characters, except for slash and the null byte.
- 67 (7) The purpose of POSIX.1 is to standardize the common, existing definition
68 (see “Application Oriented” in the Introduction) of the UNIX system pro-
69 gramming interface, not to change it. Mandating case insensitivity
70 would make all historical implementations nonstandard.
- 71 (8) Not only the interface, but also application programs would need to
72 change, counter to the purpose of having minimal changes to existing
73 application code.
- 74 (9) At least one of the original developers of the UNIX system has expressed
75 objection in the strongest terms to either requiring case insensitivity or
76 making it an option, mostly on the basis that POSIX.1 should not hinder
77 portability of application programs across related implementations in
78 order to allow compatibility with unrelated operating systems.

9 Two proposals were entertained regarding case folding in filenames:

- 10 — Remove all wording that previously permitted case folding.
11 Rationale: Case folding is inconsistent with portable filename character set
12 definition and filename definition (all characters except slash and null). No
13 known implementations allowing all characters except slash and null also
14 do case folding.
- 15 — Change “though this practice is not recommended:” to “although this prac-
16 tice is strongly discouraged.”
17 Rationale: If case folding must be included in POSIX.1, the wording should
18 be stronger to discourage the practice.

9 The consensus selected the first proposal. Otherwise, a portable application
0 would have to assume that case folding would occur when it was not wanted, but
1 that it would not occur when it was wanted.

2 **B.2.3.6 file times update:** This subclause reflects the actions of historical
3 implementations. The times are not updated immediately, but are only marked
4 for update by the functions. An implementation may update these times
5 immediately.

1096 The accuracy of the time update values is intentionally left unspecified so that
1097 systems can control the bandwidth of a possible covert channel.

1098 The wording was carefully chosen to make it clear that there is no requirement
1099 that the conformance document contain information that might incidentally affect
1100 file update times. Any function that performs pathname resolution might update
1101 several *st_atime* fields. Functions such as *getpwnam()* and *getgrnam()* might
1102 update the *st_atime* field of some specific file or files. It is intended that these are
1103 not required to be documented in the conformance document, but they should
1104 appear in the system documentation.

1105 **B.2.3.7 pathname resolution:** What the filename dot-dot refers to relative to
1106 the root directory is implementation defined. In Version 7 it refers to the root
1107 directory itself; this is the behavior mentioned in the standard. In some
1108 networked systems the construction *../hostname/* is used to refer to the root
1109 directory of another host, and POSIX.1 permits this behavior.

1110 Other networked systems use the construct *//hostname* for the same purpose;
1111 i.e., a double initial slash is used. There is a potential problem with existing
1112 applications that create full pathnames by taking a trunk and a relative path-
1113 name and making them into a single string separated by */*, because they can
1114 accidentally create networked pathnames when the trunk is */*. This practice is
1115 not prohibited because such applications can be made to conform by simply
1116 changing to use *//* as a separator instead of */*:

- 1117 (1) If the trunk is */*, the full path name will begin with *///* (the initial */* and
1118 the separator *//*). This is the same as */*, which is what is desired. (This
1119 is the general case of making a relative pathname into an absolute one by
1120 prefixing with *///* instead of */*.)
- 1121 (2) If the trunk is */A*, the result is */A//...*; since nonleading sequences of
1122 two or more slashes are treated as a single slash, this is equivalent to the
1123 desired */A/...*
- 1124 (3) If the trunk is *//A*, the implementation-defined semantics will apply.
1125 (The multiple slash rule would apply.)

1126 Application developers should avoid generating pathnames that start with *///*.
1127 Implementations are strongly encouraged to avoid using this special interpreta-
1128 tion since a number of applications currently do not follow this practice and may
1129 inadvertently generate *///...*.

1130 The term root directory is only defined in POSIX.1 relative to the process. In some
1131 implementations, there may be no absolute root directory. The initialization of
1132 the root directory of a process is implementation defined.

1133 B.2.4 Error Numbers

1134 The definition of *errno* in POSIX.1 is stricter than that in the C Standard [2]. The
1135 C Standard [2] merely requires that it be an assignable *lvalue*. The POSIX.1
1136 *extern int errno* meets that requirement and supports historical usage as well.

Checking the value of *errno* alone is not sufficient to determine the existence or type of an error, since it is not required that a successful function call clear *errno*. The variable *errno* should only be examined when the return value of a function indicates that the value of *errno* is meaningful. In that case, the function is required to set the variable to something other than zero.

A successful function call may set the value of *errno* to zero, or to any other value (except where specifically prohibited; see B.5.4.1). But it is meaningless to do so, since the value of *errno* is undefined except when the description of a function explicitly states that it is set, and no function description states that it should be set on a successful call. Most functions in most implementations do not change *errno* on successful completion. Exceptions are *isatty()* and *ptrace()*. The latter is not in POSIX.1, but is widely implemented and clears *errno* when called. The value of *errno* is not defined unless all signal handlers that use functions that could change *errno* save and restore it.

POSIX.1 requires (in the Errors subclauses of function descriptions) certain error values to be set in certain conditions because many existing applications depend on them. Some error numbers, such as [EFAULT], are entirely implementation defined and are noted as such in their description in 2.4. This subclause otherwise allows wide latitude to the implementation in handling error reporting.

Some of the Errors clauses in POSIX.1 have two subclauses. The first:

“If any of the following conditions occur, the *foo()* function shall return *-1* and set *errno* to the corresponding value:”

could be called the “mandatory” subclause. The second:

“For each of the following conditions, when the condition is detected, the *foo()* function shall return *-1* and set *errno* to the corresponding value:”

could be informally known as the “optional” subclause. This latter subclause has evolved in meaning over time. In early drafts, it was only used for error conditions that could not be detected by certain hardware configurations, such as the [EFAULT] error, as described below. The subclause recently has also added conditions associated with optional system behavior, such as job control errors. Attempting to infer the quality of an implementation based on whether it detects such conditions is not useful.

Following each one-word symbolic name for an error, there is a one-line tag, which is followed by a description of the error. The one-line tag is merely a mnemonic or historical referent and is not part of the specification of the error. Many programs print these tags on the standard error stream [often by using the C Standard {2} *perror()* function] when the corresponding errors are detected, but POSIX.1 does not require this action.

[EFAULT]	Most historical implementations do not catch an error and set <i>errno</i> when an invalid address is given to the functions <i>wait()</i> , <i>time()</i> , or <i>times()</i> . Some implementations cannot reliably detect an invalid address. And most systems that detect invalid addresses will do so only for a system call, not for a library routine.
----------	---

1182	[EINTR]	POSIX.1 prohibits conforming implementations from restarting interrupted system calls. However, it does not require that
1183		[EINTR] be returned when another legitimate value may be
1184		substituted; e.g., a partial transfer count when <i>read()</i> or <i>write()</i>
1185		are interrupted. This is only given when the signal catching
1186		function returns normally as opposed to returns by mechanisms
1187		like <i>longjmp()</i> or <i>siglongjmp()</i> .
1188		
1189	[ENOMEM]	The term <i>main memory</i> is not used in POSIX.1 because it is
1190		implementation defined.
1191	[ENOTTY]	The symbolic name for this error is derived from a time when
1192		device control was done by <i>ioctl()</i> and that operation was only
1193		permitted on a terminal interface. The term “TTY” is derived
1194		from <i>teletypewriter</i> , the devices to which this error originally
1195		applied.
1196	[EPIPE]	This condition normally generates the signal SIGPIPE; the error
1197		is returned if the signal does not terminate the process.
1198	[EROFS]	In historical implementations, attempting to <i>unlink()</i> or
1199		<i>rmdir()</i> a mount point would generate an [EBUSY] error. An
1200		implementation could be envisioned where such an operation
1201		could be performed without error. In this case, if <i>either</i> the
1202		directory entry or the actual data structures reside on a read-
1203		only file system, [EROFS] is the appropriate error to generate.
1204		(For example, changing the link count of a file on a read-only
1205		file system could not be done, as is required by <i>unlink()</i> , and
1206		thus an error should be reported.)
1207	Two error numbers, [EDOM] and [ERANGE], were added to this subclause pri-	
1208	marily for consistency with the C Standard {2}.	

1209 B.2.5 Primitive System Data Types

1210 The requirement that additional types defined in this subclause end in “_t” was
 1211 prompted by the problem of namespace pollution (see B.2.7.2). It is difficult to
 1212 define a type (where that type is not one defined by POSIX.1) in one header file
 1213 and use it in another without adding symbols to the namespace of the program.
 1214 To allow implementors to provide their own types, all POSIX.1 conforming applica-
 1215 tions are required to avoid symbols ending in “_t”, which permits the implementor
 1216 to provide additional types. Because a major use of types is in the definition of
 1217 structure members, which can (and in many cases must) be added to the struc-
 1218 tures defined in POSIX.1, the need for additional types is compelling.

1219 The types such as *ushort* and *ulong*, which are in common usage, are not defined
 1220 in POSIX.1 (although *ushort_t* would be permitted as an extension). They can be
 1221 added to `<sys/types.h>` using a feature test macro (see 2.7.2). A suggested
 1222 symbol for these is `_SYSIII`. Similarly, the types like *u_short* would probably be
 1223 best controlled by `_BSD`.

1224 Some of these symbols may appear in other headers; see 2.7.

225	<i>dev_t</i>	This type may be made large enough to accommodate host-
226		locality considerations of networked systems.
227		This type must be arithmetic. Earlier drafts allowed this to be
228		nonarithmetic (such as a structure) and provided a <i>samefile()</i>
229		function for comparison.
230	<i>gid_t</i>	Some implementations had separated <i>gid_t</i> from <i>uid_t</i> before
231		POSIX.1 was completed. It would be difficult for them to
232		coalesce them when it was unnecessary. Additionally, it is
233		quite possible that user IDs might be different than group IDs
234		because the user ID might wish to span a heterogeneous net-
235		work, where the group ID might not.
236		For current implementations, the cost of having a separate
237		<i>gid_t</i> will be only lexical.
238	<i>mode_t</i>	This type was chosen so that implementations could choose the
239		appropriate integral type, and for compatibility with the
240		C Standard [2]. 4.3BSD uses <i>unsigned short</i> and the <i>SVID</i> uses
241		<i>ushort</i> , which is the same. Historically, only the low-order six-
242		teen bits are significant.
243	<i>nlink_t</i>	This type was introduced in place of <i>short</i> for <i>st_nlink</i> (see
244		5.6.1) in response to an objection that <i>short</i> was too small.
245	<i>off_t</i>	This type is used only in <i>lseek()</i> , <i>fcntl()</i> , and <i><sys/stat.h></i> .
246		Many implementations would have difficulties if it were defined
247		as anything other than <i>long</i> . Requiring an integral type limits
248		the capabilities of <i>lseek()</i> to four gigabytes. See the description
249		of <i>lread()</i> in B.6.4. Also, the C Standard [2] supplies routines
250		that use larger types: see <i>fgetpos()</i> and <i>fsetpos()</i> in B.6.5.3.
251	<i>pid_t</i>	The inclusion of this symbol was controversial because it is tied
252		to the issue of the representation of a process ID as a number.
253		From the point of view of a portable application, process IDs
254		should be “magic cookies” ²⁾ that are produced by calls such as
255		<i>fork()</i> , used by calls such as <i>waitpid()</i> or <i>kill()</i> , and not other-
256		wise analyzed (except that the sign is used as a flag for certain
257		operations).
258		The concept of a {PID_MAX} value interacted with this in early
259		drafts. Treating process IDs as an opaque type both removes
260		the requirement for {PID_MAX} and allows systems to be more
261		flexible in providing process IDs that span a large range of
262		values, or a small one.

263 2) An historical term meaning: “An opaque object, or token, of determinate size, whose significance
264 is known only to the entity which created it. An entity receiving such a token from the
265 generating entity may only make such use of the ‘cookie’ as is defined and permitted by the
266 supplying entity.”

1267 Since the values in *uid_t*, *gid_t*, and *pid_t* will be numbers gen-
 1268 erally, and potentially both large in magnitude and sparse,
 1269 applications that are based on arrays of objects of this type are
 1270 unlikely to be fully portable in any case. Solutions that treat
 1271 them as magic cookies will be portable.

1272 {CHILD_MAX} precludes the possibility of a “toy implementa-
 1273 tion,” where there would only be one process.

1274 *ssize_t* This is intended to be a signed analog of *size_t*. The wording is
 1275 such that an implementation may either choose to use a longer
 1276 type or simply to use the signed version of the type that under-
 1277 lies *size_t*. All functions that return *ssize_t* [*read()* and *write()*]
 1278 describe as “implementation defined” the result of an input
 1279 exceeding {SSIZE_MAX}. It is recognized that some implemen-
 1280 tations might have *ints* that are smaller than *size_t*. A port-
 1281 able application would be constrained not to perform I/O in
 1282 pieces larger than {SSIZE_MAX}, but a portable application
 1283 using extensions would be able to use the full range if the
 1284 implementation provided an extended range, while still having
 1285 a single type-compatible interface.

1286 The symbols *size_t* and *ssize_t* are also required in
 1287 `<unistd.h>` to minimize the changes needed for calls to *read()*
 1288 and *write()*. Implementors are reminded that it must be possi-
 1289 ble to include both `<sys/types.h>` and `<unistd.h>` in the
 1290 same program (in either order) without error.

1291 *uid_t* Before the addition of this type, the data types used to
 1292 represent these values varied throughout early drafts. The
 1293 `<sys/stat.h>` header defined these values as type *short*, the
 1294 `<passwd.h>` file (now `<pwd.h>` and `<grp.h>`) used an *int*,
 1295 and *getuid()* returned an *int*. In response to a strong objection
 1296 to the inconsistent definitions, all the types were switched to
 1297 *uid_t*.

1298 In practice, those historical implementations that use varying
 1299 types of this sort can typedef *uid_t* to *short* with no serious
 1300 consequences.

1301 The problem associated with this change concerns object com-
 1302 patibility after structure size changes. Since most implementa-
 1303 tions will define *uid_t* as a short, the only substantive change
 1304 will be a reduction in the size of the *passwd* structure. Conse-
 1305 quently, implementations with an overriding concern for object
 1306 compatibility can pad the structure back to its current size. For
 1307 that reason, this problem was not considered critical enough to
 1308 warrant the addition of a separate type to POSIX.1.

1309 The types *uid_t* and *gid_t* are magic cookies. There is no
 1310 {UID_MAX} defined by POSIX.1, and no structure imposed on
 1311 *uid_t* and *gid_t* other than that they be positive arithmetic
 1312 types. (In fact, they could be *unsigned char*.) There is no max-
 1313 imum or minimum specified for the number of distinct user or
 1314 group IDs.

15 B.2.6 Environment Description

16 The variable *environ* is not intended to be declared in any header, but rather to be
17 declared by the user for accessing the array of strings that is the environment.
18 This is the traditional usage of the symbol. Putting it into a header could break
19 some programs that use the symbol for their own purposes.

20 **LC_*** The description of the environment variable names starting
21 with the characters “LC_” acknowledges the fact that the inter-
22 faces presented in the current version of POSIX.1 are not com-
23 plete and may be extended as new international functionality is
24 required. In the C Standard [2], names preceded by “LC_” are
25 reserved in the name space for future categories.

26 To avoid name clashes, new categories and environment vari-
27 ables are divided into two classifications: implementation
28 independent and implementation dependent.

29 Implementation-independent names will have the following
30 format:

31 **LC_NAME**

32 where *NAME* is the name of the new category and environment
33 variable. Capital letters must be used for implementation-
34 independent names.

35 Implementation-dependent names must be in lowercase letters,
36 as below:

37 **LC_name**

38 **PATH** Many historical implementations of the Bourne shell do not
39 interpret a trailing colon to represent the current working
40 directory and are thus nonconforming. The C Shell and the
41 KornShell conform to POSIX.1 on this point. The usual name of
42 dot may also be used to refer to the current working directory.

43 **TZ** See 8.1.1 for an explanation of the format.

44 **LOGNAME** 4.3BSD uses the environment variable **USER** for this purpose.
45 In most implementations, the value of such a variable is easily
46 forged, so security-critical applications should rely on other
47 means of determining user identity. **LOGNAME** is required to
48 be constructed from the portable filename character set for rea-
49 sons of interchange. No diagnostic condition is specified for
50 violating this rule, and no requirement for enforcement exists.
51 The intent of the requirement is that if extended characters are
52 used, the “guarantee” of portability implied by a standard is
53 voided. (See also B.2.2.2.)

54 The following environment variables have been used historically as indicated.
55 However, such use was either so variant as to not be amenable to standardization
56 or to be relevant only to other facilities not specified in POSIX.1, and they have
57 therefore been excluded. They may or may not be included in future POSIX stan-
58 dards. Until then, writers of conforming applications should be aware that

1359 details of the use of these variables are likely to vary in different contexts.

1360	IFS	Characters used as field separators.
1361	MAIL	System mailer information.
1362	PS1	Prompting string for interactive programs.
1363	PS2	Prompting string for interactive programs.
1364	SHELL	The shell command interpreter name.

1365 **B.2.7 C Language Definitions**

1366 The construct `<name.h>` for headers is also taken from the C Standard [2].

1367 **B.2.7.1 Symbols From the C Standard**

1368 The reservation of identifiers is paraphrased from the C Standard [2]. The text is
 1369 included because it needs to be part of POSIX.1, regardless of possible changes in
 1370 future versions of the C Standard [2]. The reservation of other namespaces is par-
 1371 ticularly for `<errno.h>`.

1372 These identifiers may be used by implementations, particularly for feature test
 1373 macros. Implementations should not use feature test macro names that might be
 1374 reasonably used by a standard.

1375 The requirement for representing the number of clock ticks in 24 h refers to the
 1376 interval defined by POSIX.1, not to the interval defined by the C Standard [2].

1377 Including headers more than once is a reasonably common practice, and it should
 1378 be carried forward from the C Standard [2]. More significantly, having definitions
 1379 in more than one header is explicitly permitted. Where the potential declaration
 1380 is “benign” (the same definition twice) the declaration can be repeated, if that is
 1381 permitted by the compiler. (This is usually true of macros, for example.) In those
 1382 situations where a repetition is not benign (e.g., typedefs), conditional compilation
 1383 must be used. The situation actually occurs both within the C Standard [2] and
 1384 within POSIX.1: `time_t` should be in `<sys/types.h>`, and the C Standard [2]
 1385 mandates that it be in `<time.h>`. POSIX.1 requires using `<sys/types.h>` with
 1386 `<time.h>` because of the common-usage environment.

1387 **B.2.7.2 POSIX.1 Symbols**

1388 This subclause addresses the issue of “namespace pollution.” The C Standard [2]
 1389 requires that the namespace beyond what it reserves not be altered except by
 1390 explicit action of the application writer. This subclause defines the actions to add
 1391 the POSIX.1 symbols for those headers where both the C Standard [2] and POSIX.1
 1392 need to define symbols. Where there are nonoverlapping uses of headers, there is
 1393 no problem.

1394 The list of symbols defined in the C Standard [2] is summarized in the rationale
 1395 associated with Annex C.

1396 Implementors should note that the requirement on type conversion disallows

using an older declaration as a prototype and in effect requires that the number of arguments in the prototype match that given in POSIX.1.

When headers are used to provide symbols, there is a potential for introducing symbols that the application writer cannot predict. Ideally, each header should only contain one set of symbols, but this is not practical for historical reasons. Thus, the concept of feature test macros is included. This is done in a general manner because it is expected that future additions to POSIX.1 and other related standards will have this same problem. (Future standards not constrained by historical practice should avoid the problem by using new header files rather than using ones already extant.)

This idea is split into two subclauses: 2.7.2.1 covers the case of the C Standard {2} conformant systems, where the requirements of the C Standard {2} are that unless specifically requested the application will not see any other symbols, and “Common Usage,” where the default set of symbols is not well controlled and backwards compatibility is an issue.

The common usage case is the more difficult to define. In the C Standard {2} case, each feature test macro simply adds to the possible symbols. In common usage, `_POSIX_SOURCE` is a special case in that it reduces the set to the sum of the C Standard {2} and POSIX.1. (The developers of the C Standard {2} will determine if they want a similar macro to limit the features to just the C Standard {2}; the wording permits this because under those circumstances `_POSIX_SOURCE` would be just another ordinary feature test macro. The only order requirement is “before headers.”)

If `_POSIX_SOURCE` is not defined in a common-usage environment, the user presumably gets the same results as in previous releases. Some applications may today be conformant without change, so they would continue to compile as long as common usage is provided. When the C Standard {2} is the default they will have to change (unless they are already C Standard {2} conformant), but this can be done gradually.

Note that the net result of defining `_POSIX_SOURCE` at the beginning of a program is in either case the same: the implementation-defined symbols are only visible if they are requested. (But if `_POSIX_SOURCE` is not used, the implementation default, which is probably backwards compatible, determines their visibility.)

The area of namespace pollution versus additions to structures is difficult because of the macro structure of C. The following discussion summarizes all the various problems with and objections to the issue.

Note the phrase “user defined macro.” Users are not permitted to define macro names (or any other name) beginning with `_[A-Z_]`. Thus, the conflict cannot occur for symbols reserved to the vendor’s namespace, and the permission to add fields automatically applies, without qualification, to those symbols.

- (1) Data structures (and unions) need to be defined in headers by implementations to meet certain requirements of POSIX.1 and the C Standard {2}.
- (2) The structures defined by POSIX.1 are typically minimal, and any practical implementation would wish to add fields to these structures either to hold additional related information or for backwards compatibility (or

1443 both). Future standards (and de facto standards) would also wish to add
 1444 to these structures. Issues of field alignment make it impractical (at
 1445 least in the general case) to simply omit fields when they are not defined
 1446 by the particular standard involved.

1447 Struct *dirent* is an example of such a minimal structure (although one
 1448 could argue about whether the other fields need visible names). The
 1449 *st_rdev* field of most implementations' *stat* structure is a common exam-
 1450 ple where extension is needed and where a conflict could occur.

1451 (3) Fields in structures are in an independent namespace, so the addition of
 1452 such fields presents no problem to the C language itself in that such
 1453 names cannot interact with identically named user symbols because
 1454 access is qualified by the specific structure name.

1455 (4) There is an exception to this: macro processing is done at a lexical level.
 1456 Thus, symbols added to a structure might be recognized as user-provided
 1457 macro names at the location where the structure is declared. This only
 1458 can occur if the user-provided name is declared as a macro before the
 1459 header declaring the structure is included. The user's use of the name
 1460 after the declaration cannot interfere with the structure because the sym-
 1461 bol is hidden and only accessible through access to the structure.
 1462 Presumably, the user would not declare such a macro if there was an
 1463 intention to use that field name.

1464 (5) Macros from the same or a related header might use the additional fields
 1465 in the structure, and those field names might also collide with user mac-
 1466 ros. Although this is a less frequent occurrence, since macros are
 1467 expanded at the point of use, no constraint on the order of use of names
 1468 can apply.

1469 (6) An "obvious" solution of using names in the reserved namespace and
 1470 then redefining them as macros when they should be visible does not
 1471 work because this has the effect of exporting the symbol into the general
 1472 namespace. For example, given a (hypothetical) system-provided header
 1473 `<h.h>`, and two parts of a C program in `a.c` and `b.c`:

1474 In header `<h.h>`:

```

1475     struct foo {
1476         int __i;
1477     }
1478
1479     #ifdef _FEATURE_TEST
1480     #define i __i;
1481     #endif
  
```

1481 In file `a.c`:

```

1482     #include h.h
1483     extern int i;
1484     ...
  
```

1485 In file `b.c`:

```

36     extern int i;
37     ...

```

38 The symbol that the user thinks of as `i` in both files has an external
39 name of "`__i`" in `a.c`; the same symbol `i` in `b.c` has an external name
40 "`i`" (ignoring any hidden manipulations the compiler might perform on
41 the names). This would cause a mysterious name resolution problem
42 when `a.o` and `b.o` are linked.

33 Simply avoiding definition then causes alignment problems in the
34 structure.

35 A structure of the form

```

36     struct foo {
37         union {
38             int __i;
39 #ifdef _FEATURE_TEST
40             int i;
41 #endif
42         } __ii;
43     }

```

44 does not work because the name of the logical field `i` is "`__ii.i`", and
45 introduction of a macro to restore the logical name immediately reintro-
46 duces the problem discussed previously (although its manifestation
47 might be more immediate because a syntax error would result if a recur-
48 sive macro did not cause it to fail first).

09 (7) A more workable solution would be to declare the structure:

```

10     struct foo {
11 #ifdef _FEATURE_TEST
12         int i;
13 #else
14         int __i;
15 #endif
16     }

```

17 However, if a macro (particularly one required by a standard) is to be
18 defined that uses this field, two must be defined: one that uses `i`, the
19 other that uses `__i`. If more than one additional field is used in a macro
20 and they are conditional on distinct combinations of features, the com-
21 plexity goes up as 2^n .

22 All this leaves a difficult situation: vendors must provide very complex headers to
23 deal with what is conceptually simple and safe: adding a field to a structure. It is
24 the possibility of user-provided macros with the same name that makes this
25 difficult.

26 Several alternatives were proposed that involved constraining the user's access to
27 part of the namespace available to the user (as specified by the C Standard {2}).
28 In some cases, this was only until all the headers had been included. There were
29 two proposals discussed that failed to achieve consensus:

1530 — Limiting it for the whole program.

1531 — Restricting the use of identifiers containing only uppercase letters until
1532 after all system headers had been included. It was also pointed out that
1533 because macros might wish to access fields of a structure (and macro
1534 expansion occurs totally at point of use) restricting names in this way
1535 would not protect the macro expansion, and thus the solution was
1536 inadequate.

1537 It was finally decided that reservation of symbols would occur, but as constrained.

1538 The current wording also allows the addition of fields to a structure, but requires
1539 that user macros of the same name not interfere. This allows vendors to either:

1540 — Not create the situation [do not extend the structures with user-accessible
1541 names or use the solution in (7) above] or

1542 — Extend their compilers to allow some way of adding names to structures
1543 and macros safely.

1544 There are at least two ways that the compiler might be extended: add new
1545 preprocessor directives that turn off and on macro expansion for certain symbols
1546 (without changing the value of the macro) and a function or lexical operation that
1547 suppresses expansion of a word. The latter seems more flexible, particularly
1548 because it addresses the problem in macros as well as in declarations.

1549 The following seems to be a possible implementation extension to the C language
1550 that will do this: any token that during macro expansion is found to be preceded
1551 by three # symbols shall not be further expanded in exactly the same way as
1552 described for macros that expand to their own name as in section 3.8.3.4 of the
1553 C Standard {2}. A vendor may also wish to implement this as an operation that is
1554 lexically a function, which might be implemented as

1555 `#define __safe_name(x) ###x`

1556 Using a function notation would insulate vendors from changes in standards until
1557 such a functionality is standardized (if ever). Standardization of such a function
1558 would be valuable because it would then permit third parties to take advantage of
1559 it portably in software they may supply.

1560 The symbols that are “explicitly permitted, but not required by this part of
1561 ISO/IEC 9945” include those classified below. (That is, the symbols classified
1562 below might, but are not required to, be present when `_POSIX_SOURCE` is
1563 defined.)

1564 — Symbols in 2.8 and 2.9 that are defined to indicate support for options or
1565 limits that are constant at compile-time.

1566 — Symbols in the namespace reserved for the implementation by the
1567 C Standard {2}.

1568 — Symbols in a namespace reserved for a particular type of extension (e.g.,
1569 type names ending with `_t` in `<sys/types.h>`).

1570 — Additional members of structures or unions whose names do not reduce the
1571 namespace reserved for applications (see B.2.7.2).

The phrase “when that header is included” was chosen to allow any fine structure of auxiliary headers the implementor may choose to use, as long as the net result is as required.

There are several common environments available today where a feature test macro would be useful to applications programmers during the transition to standard-conforming environments from certain common historical environments. The symbols in Table B-1, derived from common porting bases and industry specifications are suggested.

Table B-1 – Suggested Feature Test Macros

Symbol	Description
_V7	Version 7
_BSD	General BSD systems
_BSD4_2	4.2BSD
_BSD4_3	4.3BSD
_SYSIII	System III
_SYSV	System V.1, V.2
_SYSV3	System V.3
_XPGn	X/Open Portability Guide, Issue n
_USR_GROUP	The 1984 /usr/group standard

Only symbols that are actually in the porting base or industry specification should be enabled by these symbols.

Feature test macros for implementation extensions will also probably be required. Quite a few of these are traditionally available, but are in violation of the intent of namespace pollution control. These can be made conforming simply by prefixing them with an underscore. Symbols beginning with “_POSIX” are strongly discouraged, as they will probably be used by later revisions of POSIX.1.

The environment for compilation has traditionally been fairly portable in historical systems, but during the transition to the C Standard [2] there will be confusion about how to specify that a C Standard [2] compiler is expected, as considerations of backwards compatibility will constrain many implementors from providing a conformant environment replacing the traditional one. This concern has more to do with the issues of namespace than with the syntax of the language accepted, which is highly compatible.

For systems that are sufficiently similar to traditional UNIX systems for this to make sense, it is suggested that if a compilation line of the form

```
cc -D__STDC__ ...
```

is provided, that the system provide an environment that is conformant with the C Standard [2], at least with respect to namespace.

It was decided to use feature test macros, rather than the inclusion of a header, both because <unistd.h> was already in use and would itself have this problem, and because the underlying mechanism would probably have been this anyway, but in a less flexible fashion.

1616 POSIX.1 requires that headers be included in all cases, although it is not directly
 1617 clear from the text at this point in the standard. If a function does not need any
 1618 special types, then it must be declared in `<unistd.h>`, as stated here. If it does
 1619 require something special, then it has an associated header, and the program will
 1620 not compile without that header.

1621 **B.2.7.3 Headers and Function Prototypes**

1622 The statement that names need not be carried forward literally exists for several
 1623 reasons. These include the fact that some vendors may historically use other
 1624 names and that the names are irrelevant to application portability. More impor-
 1625 tantly, because of the pervasive nature of C macros, a declaration of the form:

```
1626     kill (pid_t pid, int sig);
```

1627 could be seriously undermined by a (perfectly valid) user declaration of the form:

```
1628     #define pid statusstruct.pidinfo
```

1629 **B.2.8 Numerical Limits**

1630 This subclause clarifies the scope and mutability of several classes of limits.

1631 **B.2.8.1 C Language Limits**

1632 See also 2.7 and B.1.1.1.

1633 {CHAR_MIN} It is possible to tell if the implementation supports native char-
 1634 acter comparison as signed or unsigned by comparing this limit
 1635 to zero.

1636 {WORD_BIT} This limit has been omitted, as it is not referenced elsewhere in
 1637 POSIX.1.

1638 No limits are given in `<limits.h>` for floating point values because none of the
 1639 functions in POSIX.1 use floating point values, and all the functions that do that
 1640 are imported from the C Standard [2] by 8.1, as are the limits that apply to the
 1641 floating point values associated with them.

1642 Though limits to the addresses to system calls were proposed, they were not
 1643 included in POSIX.1 because it is not clear how to implement them for the range of
 1644 systems being considered, and no complete proposal was ever received. Limits
 1645 regarding hardware register characteristics were similarly proposed and not
 1646 attempted.

1647 **B.2.8.2 Minimum Values**

1648 There has been confusion about the minimum maxima, and when that is under-
 1649 stood there is still a concern about providing ways to allocate storage based on the
 1650 symbols. This is particularly true for those in 2.8.4 where an indeterminate value
 1651 will leave the programmer with no symbol upon which to fall back.

1652 Providing explicit symbols for the minima (from the implementor's point of view,
 1653 or maxima from the the application's point of view) helps to resolve possible

confusion. Symbols are still provided for the actual value, and it is expected that many applications will take advantage of these larger values, but they need not do so unless it is to their advantage. Where the values in this subclause are adequate for the application, it should use them. These are given symbolically both because it is easier to understand and because the values of these symbols could change between revisions of POSIX.1. Arguments to “good programming practice” also apply.

B.2.8.3 Run-Time Increaseable Values

The heading of the far-right column of the table is given as “Minimum Value” rather than “Value” in order to emphasize that the numbers given in that column are minimal for the actual values a specific implementation is permitted to define in its `<limits.h>`. The values in the actual `<limits.h>` define, in turn, the maximum amount of a given resource that a Conforming POSIX.1 Application can depend on finding when translated to execute on that implementation. A Conforming POSIX.1 Application Using Extensions must function correctly even if the value given in `<limits.h>` is the minimum that is specified in POSIX.1. (The application may still be written so that it performs more efficiently when a larger value is found in `<limits.h>`.) A conforming implementation must provide at least as much of a particular resource as that given by the value in POSIX.1. An implementation that cannot meet this requirement (a “toy implementation”) cannot be a conforming implementation.

B.2.8.4 Run-Time Invariant Values (Possibly Indeterminate)

{CHILD_MAX} This name can be misleading. This limit applies to all processes in the system with the same user ID, regardless of ancestry.

B.2.8.5 Pathname Variable Values

{MAX_INPUT} Since the only use of this limit is in relation to terminal input queues, it mentions them specifically. This limit was originally named {MAX_CHAR}. Application writers should use {MAX_INPUT} primarily as an indication of the number of bytes that can be written as a single unit by one Conforming POSIX.1 Application Using Extensions communicating with another via a terminal device. It is not implied that input lines received from terminal devices always contain {MAX_INPUT} bytes or fewer: an application that attempts to read more than {MAX_INPUT} bytes from a terminal may receive more than {MAX_INPUT} bytes.

It is not obvious that {MAX_INPUT} is of direct value to the application writer. The existence of such a value (whatever it may be) is directly of use in understanding how the tty driver works (particularly with respect to flow control and dropped characters). The value can be determined by finding out when flow control takes effect (see the description of IXOFF in 7.1.2.2).

1698 Understanding that the limit exists and knowing its magnitude
 1699 is important to making certain classes of applications work
 1700 correctly. It is unlikely to be used in an application, but its
 1701 presence makes POSIX.1 clearer.

1702 {PATH_MAX} A Conforming POSIX.1 Application or Conforming POSIX.1
 1703 Application Using Extensions that, for example, compiles to use
 1704 different algorithms depending on the value of {PATH_MAX}
 1705 should use code such as:

```

1706 #if defined(PATH_MAX) && PATH_MAX < 512
1707     ...
1708 #else
1709 #if defined(PATH_MAX) /* PATH_MAX >= 512 */
1710     ...
1711 #else /* PATH_MAX indeterminate */
1712     ...
1713 #endif
1714 #endif
  
```

1715 This is because the value tends to be very large or indeter-
 1716minate on most historical implementations (it is arbitrarily
 1717large on System V). On such systems there is no way to quan-
 1718tify the limit, and it seems counterproductive to include an
 1719artificially small fixed value in `<limits.h>` in such cases.

1720 B.2.9 Symbolic Constants

1721 B.2.9.1 Symbolic Constants for the *access()* Function

1722 There is no additional rationale provided for this subclause.

1723 B.2.9.2 Symbolic Constants for the *lseek()* Function

1724 There is no additional rationale provided for this subclause.

1725 B.2.9.3 Compile-Time Symbolic Constants for Portability Specifications

1726 The purpose of this material is to allow an application developer to have a chance
 1727 to determine whether a given application would run (or run well) on a given
 1728 implementation. To this purpose has been added that of simplifying development
 1729 of verification suites for POSIX.1. The constants given here were originally pro-
 1730 posed for a separate file, `<posix.h>`, but it was decided that they should appear
 1731 in `<unistd.h>` along with other symbolic constants.

1732 B.2.9.4 Execution-Time Symbolic Constants for Portability Specifications

1733 Without the addition of `{_POSIX_NO_TRUNC}` and `{_PC_NO_TRUNC}` to this list,
 1734 POSIX.1 says nothing about the effect of a pathname component longer than
 1735 `{NAME_MAX}`. There are only two effects in common use in implementations:
 1736 truncation or an error. It is desirable to limit allowable behavior to these two

cases. It is also desirable to permit applications to determine what an implementation's behavior is because services that are available with one behavior may be impractical to provide with the other. However, since the behavior may vary from one file system to another, it may be necessary to use *pathconf()* to resolve it.

B.3 Process Primitives

Consideration was given to enumerating all characteristics of a process defined by POSIX.1 and describing each function in terms of its effects on those characteristics, rather than English text. This is quite different from any known descriptions of historical implementations, and it was not certain that this could be done adequately and completely enough to produce a usable standard. Providing such descriptions in addition to the text was also considered. This was not done because it would provide at best two redundant descriptions, and more likely two descriptions with subtle inconsistencies.

B.3.1 Process Creation and Execution

Running a new program takes two steps. First the existing process (the parent) calls the *fork()* function, producing a new process (the child), which is a copy of itself. One of these processes (normally, but not necessarily, the child) then calls one of the *exec* functions to overlay itself with a copy of the new process image.

If the new program is to be run synchronously (the parent suspends execution until the child completes), the parent process then uses either the *wait()* or *waitpid()* function. If the new program is to be run asynchronously, it does not suffice to simply omit the *wait()* or *waitpid()* call, because after the child terminates it continues to hold some resources until it is waited for. A common way to produce ("spawn") a descendant process that does not need to be waited on is to *fork()* to produce a child and *wait()* on the child. The child *fork()*s again to produce a grandchild. The child then exits and the parent's *wait()* returns. The grandchild is thus disinherited by its grandparent.

A simpler method (from the programmer's point of view) of spawning is to do

```
system("something &");
```

However, this depends on features of a process (the shell) that are outside the scope of POSIX.1, although they are currently being addressed by the working group preparing ISO/IEC 9945-2 (B36).

B.3.1.1 Process Creation

Many historical implementations have timing windows where a signal sent to a process group (e.g., an interactive SIGINT) just prior to or during execution of *fork()* is delivered to the parent following the *fork()* but not to the child because the *fork()* code clears the child's set of pending signals. POSIX.1 does not require, or even permit, this behavior. However, it is pragmatic to expect that problems of this nature may continue to exist in implementations that appear to conform to

1777 POSIX.1 and pass available verification suites. This behavior is only a conse-
 1778 quence of the implementation failing to make the interval between signal genera-
 1779 tion and delivery totally invisible. From the application's perspective, a *fork()* call
 1780 should appear atomic. A signal that is generated prior to the *fork()* should be
 1781 delivered prior to the *fork()*. A signal sent to the process group after the *fork()*
 1782 should be delivered to both parent and child. The implementation might actually
 1783 initialize internal data structures corresponding to the child's set of pending sig-
 1784 nals to include signals sent to the process group during the *fork()*. Since the
 1785 *fork()* call can be considered as atomic from the application's perspective, the set
 1786 would be initialized as empty and such signals would have arrived after the
 1787 *fork()*. See also B.3.3.1.2.

1788 One approach that has been suggested to address the problem of signal inheri-
 1789 tance across *fork()* is to add an [EINTR] error, which would be returned when a
 1790 signal is detected during the call. While this is preferable to losing signals, it was
 1791 not considered an optimal solution. Although it is not recommended for this pur-
 1792 pose, such an error would be an allowable extension for an implementation.

1793 The [ENOMEM] error value is reserved for those implementations that detect and
 1794 distinguish such a condition. This condition occurs when an implementation
 1795 detects that there is not enough memory to create the process. This is intended to
 1796 be returned when [EAGAIN] is inappropriate because there can never be enough
 1797 memory (either primary or secondary storage) to perform the operation. Because
 1798 *fork()* duplicates an existing process, this must be a condition where there is
 1799 sufficient memory for one such process, but not for two. Many historical imple-
 1800 mentations actually return [ENOMEM] due to temporary lack of memory, a case
 1801 that is not generally distinct from [EAGAIN] from the perspective of a portable
 1802 application.

1803 Part of the reason for including the optional error [ENOMEM] is because the *SVID*
 1804 {B39} specifies it and it should be reserved for the error condition specified there.
 1805 The condition is not applicable on many implementations.

1806 IEEE Std 1003.1-1988 neglected to require concurrent execution of the parent and
 1807 child of *fork()*. A system that single-threads processes was clearly not intended
 1808 and is considered an unacceptable, "toy implementation" of POSIX.1. The only
 1809 objection anticipated to the phrase "executing independently" is testability, but
 1810 this assertion should be testable. Such tests require that both the parent and
 1811 child can block on a detectable action of the other, such as a write to a pipe or a
 1812 signal. An interactive exchange of such actions should be possible for the system
 1813 to conform to the intent of POSIX.1.

1814 The [EAGAIN] error exists to warn applications that such a condition might occur.
 1815 Whether it will occur or not is not in any practical sense under the control of the
 1816 application because the condition is usually a consequence of the user's use of the
 1817 system, not of the application's code. Thus, no application can or should rely upon
 1818 its occurrence under any circumstances, nor should the exact semantics of what
 1819 concept of "user" is used be of concern to the application writer. Validation writ-
 1820 ers should be cognizant of this limitation.

21 **B.3.1.2 Execute a File**

22 Early drafts of POSIX.1 required that the value of *argc* passed to *main()* be “one or
23 greater.” This was driven by the same requirement in drafts of the C Standard
24 [2]. In fact, historical implementations have passed a value of zero when no argu-
25 ments are supplied to the caller of the *exec* functions. This requirement was
26 removed from the C Standard [2] and subsequently removed from POSIX.1 as well.
27 The POSIX.1 wording, in particular the use of the word “should,” requires a
28 Strictly Conforming POSIX.1 Application (see 1.3.3) to pass at least one argument
29 to the *exec* function, thus guaranteeing that *argc* be one or greater when invoked
30 by such an application. In fact, this is good practice, since many existing applica-
31 tions reference *argv*[0] without first checking the value of *argc*.

32 The requirement on a Strictly Conforming POSIX.1 Application also states that
33 the value passed as the first argument be a filename associated with the process
34 being started. Although some existing applications pass a pathname rather than
35 a filename in some circumstances, a filename is more generally useful, since the
36 common usage of *argv*[0] is in printing diagnostics. In some cases the filename
37 passed is not the actual filename of the file; for example, many implementations
38 of the *login* utility use a convention of prefixing a hyphen (–) to the actual
39 filename, which indicates to the command interpreter being invoked that it is a
40 “login shell.”

41 Some systems can *exec* shell scripts. This functionality is outside the scope of
42 POSIX.1, since it requires standardization of the command interpreter language of
43 the script and/or where to find a command interpreter. These fall in the domain
44 of the shell and utilities standard, currently under development as ISO/IEC 9945-2
45 [B36]. However, it is important that POSIX.1 neither require nor preclude any
46 reasonable implementation of this behavior. In particular, the description of the
47 [ENOEXEC] error is intended to permit discretion to implementations on whether
48 to give this error for shell scripts.

49 One common historical implementation is that the *execl()*, *execv()*, *execle()*, and
50 *execve()* functions return an [ENOEXEC] error for any file not recognizable as exe-
51 cutable, including a shell script. When the *execlp()* and *execvp()* functions
52 encounter such a file, they assume the file to be a shell script and invoke a known
53 command interpreter to interpret such files. These implementations of *execvp()*
54 and *execlp()* only give the [ENOEXEC] error in the rare case of a problem with the
55 command interpreter’s executable file. Because of these implementations the
56 [ENOEXEC] error is not mentioned for *execlp()* or *execvp()*, although implementa-
57 tions can still give it.

58 Another way that some historical implementations handle shell scripts is by
59 recognizing the first two bytes of the file as the character string *#!* and using the
60 remainder of the first line of the file as the name of the command interpreter to
61 execute.

62 Some implementations provide a third argument to *main()* called *envp*. This is
63 defined as a pointer to the environment. The C Standard [2] specifies invoking
64 *main()* with two arguments, so implementations must support applications writ-
65 ten this way. Since POSIX.1 defines the global variable *environ*, which is also pro-
66 vided by historical implementations and can be used anywhere *envp* could be
67 used, there is no functional need for the *envp* argument. Applications should use

1868 the *getenv()* function rather than accessing the environment directly via either
1869 *envp* or *environ*. Implementations are required to support the two-argument call-
1870 ing sequence, but this does not prohibit an implementation from supporting *envp*
1871 as an optional, third argument.

1872 POSIX.1 specifies that signals set to SIG_IGN remain set to SIG_IGN and that the
1873 process signal mask be unchanged across an *exec*. This is consistent with histori-
1874 cal implementations, and it permits some useful functionality, such as the *nohup*
1875 command. However, it should be noted that many existing applications wrongly
1876 assume that they start with certain signals set to the default action and/or
1877 unblocked. In particular, applications written with a simpler signal model that
1878 does not include blocking of signals, such as the one in the C Standard [2], may
1879 not behave properly if invoked with some signals blocked. Therefore, it is best not
1880 to block or ignore signals across *execs* without explicit reason to do so, and espe-
1881 cially not to block signals across *execs* of arbitrary (not closely co-operating)
1882 programs.

1883 If `{_POSIX_SAVED_IDS}` is defined, the *exec* functions always save the value of the
1884 effective user ID and effective group ID of the process at the completion of the
1885 *exec*, whether or not the set-user-ID or the set-group-ID bit of the process image
1886 file is set.

1887 The statement about *argv*[] and *envp*[] being constants is included to make expli-
1888 cit to future writers of language bindings that these objects are completely con-
1889 stant. Due to a limitation of the C Standard [2], it is not possible to state that
1890 idea in Standard C. Specifying two levels of *const*-qualification for the *argv*[]
1891 and *envp*[] parameters for the *exec* functions may seem to be the natural choice,
1892 given that these functions do not modify either the array of pointers or the charac-
1893 ters to which the function points, but this would disallow existing correct code.
1894 Instead, only the array of pointers is noted as constant. The table of assignment
1895 compatibility for *dst = src*, derived from the C Standard [2], summarizes the
1896 compatibility:

		<i>dst:</i>			
			<i>const</i>	<i>char</i>	<i>const</i>
		<i>char</i> *[]	<i>char</i> *[]	* <i>const</i> []	<i>char</i> * <i>const</i> []
1900	<i>src:</i>				
1901	<i>char</i> *[]	VALID		VALID	
1902	<i>const char</i> *[]		VALID		VALID
1903	<i>char</i> * <i>const</i> []			VALID	
1904	<i>const char</i> * <i>const</i> []				VALID

1905 Since all existing code has a source type matching the first row, the column that
1906 gives the most valid combinations is the third column. The only other possibility
1907 is the fourth column, but using it would require a cast on the *argv* or *envp* argu-
1908 ments. It is unfortunate that the fourth column cannot be used, because the
1909 declaration a nonexpert would naturally use would be that in the second row.

1910 The C Standard [2] and POSIX.1 do not conflict on the use of *environ*, but some
1911 historical implementations of *environ* may cause a conflict. As long as *environ* is
1912 treated in the same way as an entry point [e.g., *fork()*], it conforms to both stan-
1913 dards. A library can contain *fork()*, but if there is a user-provided *fork()*, that

914 *fork()* is given precedence and no problem ensues. The situation is similar for
915 *environ*—the POSIX.1 definition is to be used if there is no user-provided *environ*
916 to take precedence. At least three implementations are known to exist that solve
917 this problem.

918 [E2BIG] The limit {ARG_MAX} applies not just to the size of the argu-
919 ment list, but to the sum of that and the size of the environ-
920 ment list.

921 [EFAULT] Some historical systems return [EFAULT] rather than
922 [ENOEXEC] when the new process image file is corrupted. They
923 are nonconforming.

924 [ENAMETOOLONG] Since the file pathname may be constructed by taking elements
925 in the PATH variable and putting them together with the
926 filename, the [ENAMETOOLONG] condition could also be
927 reached this way.
928

929 [ETXTBSY] The error [ETXTBSY] was considered too implementation
930 dependent to include. System V returns this error when the
931 executable file is currently open for writing by some process.
932 POSIX.1 neither requires nor prohibits this behavior.

933 Other systems (such as System V) may return [EINTR] from *exec*. This is not
934 addressed by POSIX.1, but implementations may have a window between the call
935 to *exec* and the time that a signal could cause one of the *exec* calls to return with
936 [EINTR].

937 B.3.2 Process Termination

938 Early drafts drew a different distinction between normal and abnormal process
939 termination. Abnormal termination was caused only by certain signals and
940 resulted in implementation-defined “actions,” as discussed below. Subsequent
941 drafts of POSIX.1 distinguished three types of termination: normal termination
942 (as in the current POSIX.1), “simple abnormal termination,” and “abnormal termi-
943 nation with actions.” Again the distinction between the two types of abnormal
944 termination was that they were caused by different signals and that
945 implementation-defined actions would result in the latter case. Given that these
946 actions were completely implementation defined, the early drafts were only saying
947 when the actions could occur and how their occurrence could be detected, but not
948 what they were. This was of little or no use to portable applications, and thus the
949 distinction was dropped from POSIX.1.

950 The implementation-defined actions usually include, in most historical implemen-
951 tations, the creation of a file named *core* in the current working directory of the
952 process. This file contains an image of the memory of the process, together with
953 descriptive information about the process, perhaps sufficient to reconstruct the
954 state of the process at the receipt of the signal.

955 There is a potential security problem in creating a *core* file if the process was
956 set-user-ID and the current user is not the owner of the program, if the process
957 was set-group-ID and none of the user’s groups match the group of the program,

1958 or if the user does not have permission to write in the current directory. In this
1959 situation, an implementation either should not create a `core` file or should make
1960 it unreadable by the user.

1961 Despite the silence of POSIX.1 on this feature, applications are advised not to
1962 create files named `core` because of potential conflicts in many implementations.
1963 Some historical implementations use a different name than `core` for the file, such
1964 as by appending the process ID to the filename.

1965 B.3.2.1 Wait for Process Termination

1966 A call to the `wait()` or `waitpid()` function only returns status on an immediate
1967 child process of the calling process; i.e., a child that was produced by a single
1968 `fork()` call (perhaps followed by an `exec` or other function calls) from the parent. If
1969 a child produces grandchildren by further use of `fork()`, none of those grandchil-
1970 dren nor any of their descendants will affect the behavior of a `wait()` from the ori-
1971 ginal parent process. Nothing in POSIX.1 prevents an implementation from pro-
1972 viding extensions that permit a process to get status from a grandchild or any
1973 other process, but a process that does not use such extensions must be guaranteed
1974 to see status from only its direct children.

1975 The `waitpid()` function is provided for three reasons:

- 1976 — To support job control (see B.3.3).
- 1977 — To permit a nonblocking version of the `wait()` function.
- 1978 — To permit a library routine, such as `system()` or `pclose()`, to wait for its chil-
1979 dren without interfering with other terminated children for which the pro-
1980 cess has not waited.

1981 The first two of these facilities are based on the `wait3()` function provided by
1982 4.3BSD. The interface uses the `options` argument, which is identical to an argu-
1983 ment to `wait3()`. The `WUNTRACED` flag is used only in conjunction with job con-
1984 trol on systems supporting that option. Its name comes from 4.3BSD and refers to
1985 the fact that there are two types of stopped processes in that implementation:
1986 processes being traced via the `ptrace()` debugging facility and (untraced) processes
1987 stopped by job-control signals. Since `ptrace()` is not part of POSIX.1, only the
1988 second type is relevant. The name `WUNTRACED` was retained because its usage
1989 is the same, even though the name is not intuitively meaningful in this context.

1990 The third reason for the `waitpid()` function is to permit independent sections of a
1991 process to spawn and wait for children without interfering with each other. For
1992 example, the following problem occurs in developing a portable shell, or command
1993 interpreter:

```
1994     stream = popen("/bin/true");
1995     (void) system("sleep 100");
1996     (void) pclose(stream);
```

1997 On all historical implementations, the final `pclose()` will fail to reap the wait
1998 status of the `popen()`.

1999 The status values are retrieved by macros, rather than given as specific bit encod-
2000 ings as they are in most historical implementations (and thus expected by

existing programs). This was necessary to eliminate a limitation on the number of signals an implementation can support that was inherent in the traditional encodings. POSIX.1 does require that a status value of zero corresponds to a process calling `_exit(0)`, as this is the most common encoding expected by existing programs. Some of the macro names were adopted from 4.3BSD.

These macros syntactically operate on an arbitrary integer value. The behavior is undefined unless that value is one stored by a successful call to `wait()` or `waitpid()` in the location pointed to by the `stat_loc` argument. An earlier draft attempted to make this clearer by specifying each argument as `*stat_loc` rather than `stat_val`. However, that did not follow the conventions of other specifications in POSIX.1 or traditional usage. It also could have implied that the argument to the macro must literally be `*stat_loc`; in fact, that value can be stored or passed as an argument to other functions before being interpreted by these macros.

The extension that affects `wait()` and `waitpid()` and is common in historical implementations is the `ptrace()` function. It is called by a child process and causes that child to stop and return a status that appears identical to the status indicated by `WIFSTOPPED`. The status of `ptraced` children is traditionally returned regardless of the `WUNTRACED` flag [or by the `wait()` function]. Most applications do not need to concern themselves with such extensions because they have control over what extensions they or their children use. However, applications, such as command interpreters, that invoke arbitrary processes may see this behavior when those arbitrary processes misuse such extensions.

Implementations that support core file creation or other implementation-defined actions on termination of some processes traditionally provide a bit in the status returned by `wait()` to indicate that such actions have occurred.

B.3.2.2 Terminate a Process

Most C language programs should use the `exit()` function rather than `_exit()`. The `_exit()` function is defined here instead of `exit()` because the C Standard [2] defines the latter to have certain characteristics that are beyond the scope of POSIX.1, specifically the flushing of buffers on open files and the use of `atexit()`. See “The C Language” in the Introduction. There are several public-domain implementations of `atexit()` that may be of use to interface implementors who wish to incorporate it.

It is important that the consequences of process termination as described in this subclause occur regardless of whether the process called `_exit()` [perhaps indirectly through `exit()`] or instead was terminated due to a signal or for some other reason. Note that in the specific case of `exit()` this means that the `status` argument to `exit()` is treated the same as the `status` argument to `_exit()`. See also B.3.2.

A language other than C may have other termination primitives than the C language `exit()` function, and programs written in such a language should use its native termination primitives, but those should have as part of their function the behavior of `_exit()` as described in this subclause. Implementations in languages other than C are outside the scope of the present version of POSIX.1, however.

As required by the C Standard [2], using `return` from `main()` is equivalent to calling `exit()` with the same argument value. Also, reaching the end of the `main()`

2046 function is equivalent to using *exit()* with an unspecified value.

2047 A value of zero (or `EXIT_SUCCESS`, which is required by 8.1 to be zero) for the
2048 argument *status* conventionally indicates successful termination. This
2049 corresponds to the specification for *exit()* in the C Standard (2). The convention is
2050 followed by utilities such as *make* and various shells, which interpret a zero
2051 status from a child process as success. For this reason, applications should not
2052 call *exit(0)* or *_exit(0)* when they terminate unsuccessfully, for example in signal-
2053 catching functions.

2054 Historically, the implementation-dependent process that inherits children whose
2055 parents have terminated without waiting on them is called *init* and has a pro-
2056 cess ID of 1.

2057 The sending of a `SIGHUP` to the foreground process group when a controlling pro-
2058 cess terminates corresponds to somewhat different historical implementations. In
2059 System V, the kernel sends a `SIGHUP` on termination of (essentially) a controlling
2060 process. In 4.2BSD, the kernel does not send `SIGHUP` in a case like this, but the
2061 termination of a controlling process is usually noticed by a system daemon, which
2062 arranges to send a `SIGHUP` to the foreground process group with the *vhangup()*
2063 function. However, in 4.2BSD, due to the behavior of the shells that support job
2064 control, the controlling process is usually a shell with no other processes in its
2065 process group. Thus, a change to make *_exit()* behave this way in such systems
2066 should not cause problems with existing applications.

2067 The termination of a process may cause a process group to become orphaned in
2068 either of two ways. The connection of a process group to its parent(s) outside of
2069 the group depends on both the parents and their children. Thus, a process group
2070 may be orphaned by the termination of the last connecting parent process outside
2071 of the group or by the termination of the last direct descendant of the parent
2072 process(es). In either case, if the termination of a process causes a process group
2073 to become orphaned, processes within the group are disconnected from their job
2074 control shell, which no longer has any information on the existence of the process
2075 group. Stopped processes within the group would languish forever. In order to
2076 avoid this problem, newly orphaned process groups that contain stopped processes
2077 are sent a `SIGHUP` signal and a `SIGCONT` signal to indicate that they have been
2078 disconnected from their session. The `SIGHUP` signal causes the process group
2079 members to terminate unless they are catching or ignoring `SIGHUP`. Under most
2080 circumstances, all of the members of the process group are stopped if any of them
2081 are stopped.

2082 The action of sending a `SIGHUP` and a `SIGCONT` signal to members of a newly
2083 orphaned process group is similar to the action of 4.2BSD, which sends `SIGHUP`
2084 and `SIGCONT` to each stopped child of an exiting process. If such children exit in
2085 response to the `SIGHUP`, any additional descendants will receive similar treat-
2086 ment at that time. In POSIX.1, the signals will be sent to the entire process group
2087 at the same time. Also, in POSIX.1, but not in 4.2BSD, stopped processes may be
2088 orphaned, but may be members of a process group that is not orphaned; therefore,
2089 the action taken at *_exit()* must consider processes other than child processes.

2090 It is possible for a process group to be orphaned by a call to *setpgid()* or *setsid()*,
2091 as well as by process termination. POSIX.1 does not require sending `SIGHUP` and
2092 `SIGCONT` in those cases, because, unlike process termination, those cases will not

be caused accidentally by applications that are unaware of job control. An implementation can choose to send SIGHUP and SIGCONT in those cases as an extension; such an extension must be documented as required in 3.3.1.2.

B.3.3 Signals

Signals, as defined in Version 7, System III, the 1984 */usr/group Standard* (B59), and System V (except very recent releases), have shortcomings that make them unreliable for many application uses. Several objections were raised against early drafts of POSIX.1 because of this. Therefore, a new signal mechanism, based very closely on the one of 4.2BSD and 4.3BSD, was added to POSIX.1. With the exception of one feature [see item (4) below and also *sigpending()*], it is possible to implement the POSIX.1 interface as a simple library veneer on top of 4.3BSD. There are also a few minor aspects of the underlying 4.3BSD implementation (as opposed to the interface) that would also need to change to conform to POSIX.1.

The major differences from the BSD mechanism are:

- (1) *Signal mask type.* BSD uses the type *int* to represent a signal mask, thus limiting the number of signals to the number of bits in an *int* (typically 32). The new standard instead uses a defined type for signal masks. Because of this change, the interface is significantly different than it is in BSD implementations, although the functionality, and potentially the implementation, are very similar.
- (2) *Restarting system calls.* Unlike all previous historical implementations, 4.2BSD restarts some interrupted system calls rather than returning an error with *errno* set to [EINTR] after the signal-catching function returns. This change caused problems for some existing application code. 4.3BSD and other systems derived from 4.2BSD allow the application to choose whether system calls are to be restarted. POSIX.1 (in 3.3.4) does not require restart of functions because it was not clear that the semantics of system-call restart in any historical implementation were useful enough to be of value in a standard. Implementors are free to add such mechanisms as extensions.
- (3) *Signal stacks.* The 4.2BSD mechanism includes a function *sigstack()*. The 4.3BSD mechanism includes this and a function *sigreturn()*. No equivalent is included in POSIX.1 because these functions are not portable, and no sufficiently portable and useful equivalent has been identified. See also 8.3.1.
- (4) *Pending signals.* The *sigpending()* function is the sole new signal operation introduced in POSIX.1.

A proposal was considered for making reliable signals optional. However, the consensus was that this would hurt application portability, as a large percentage of applications using signals can be hurt by the unreliable aspects of historical implementations of the *signal()* mechanism defined by the C Standard [2]. This unreliability stems from the fact that the signal action is reset to SIG_DFL before the user's signal-catching routine is entered. The C Standard [2] does not require this behavior, but does explicitly permit it, and most historical implementations behave this way.

2138 For example, an application that catches the SIGINT signal using *signal()* could
2139 be terminated with no chance to recover when two such signals arrive sufficiently
2140 close in time (e.g., when an impatient user types the INTR character twice in a
2141 row on a busy system). Although the C Standard {2} no longer requires this
2142 unreliable behavior, many historical implementations, including System V, will
2143 reset the signal action to SIG_DFL. For this reason, it is strongly recommended
2144 that the *signal()* function not be used by POSIX.1 conforming applications. Imple-
2145 mentations should also consider blocking signals during the execution of the
2146 signal-catching function instead of resetting the action to SIG_DFL, but backward
2147 compatibility considerations will most likely prevent this from becoming
2148 universal.

2149 Most historical implementations do not queue signals; i.e., a process's signal
2150 handler is invoked once, even if the signal has been generated multiple times
2151 before it is delivered. A notable exception to this is SIGCLD, which, in System V,
2152 is queued. The queueing of signals is neither required nor prohibited by POSIX.1.
2153 See 3.3.1.2. It is expected that a future realtime extension to POSIX.1 will address
2154 the issue of reliable queueing of event notification.

2155 B.3.3.1 Signal Concepts

2156 B.3.3.1.1 Signal Names

2157 The restriction on the actual type used for *sigset_t* is intended to guarantee that
2158 these objects can always be assigned, have their address taken, and be passed as
2159 parameters by value. It is not intended that this type be a structure including
2160 pointers to other data structures, as that could impact the portability of applica-
2161 tions performing such operations. A reasonable implementation could be a struc-
2162 ture containing an array of some integer type.

2163 The signals described in POSIX.1 must have unique values so that they may be
2164 named as parameters of case statements in the body of a C language switch
2165 clause. However, implementation-defined signals may have values that overlap
2166 with each other or with signals specified in this document. An example of this is
2167 SIGABRT, which traditionally overlaps some other signal, such as SIGIOT.

2168 SIGKILL, SIGTERM, SIGUSR1, and SIGUSR2 are ordinarily generated only through
2169 the explicit use of the *kill()* function, although some implementations generate
2170 SIGKILL under extraordinary circumstances. SIGTERM is traditionally the
2171 default signal sent by the *kill* command.

2172 The signals SIGBUS, SIGEMT, SIGIOT, SIGTRAP, and SIGSYS were omitted from
2173 POSIX.1 because their behavior is implementation dependent and could not be
2174 adequately categorized. Conforming implementations may deliver these signals,
2175 but must document the circumstances under which they are delivered and note
2176 any restrictions concerning their delivery. The signals SIGFPE, SIGILL, and SIG-
2177 SEGV are similar in that they also generally result only from programming errors.
2178 They were included in POSIX.1 because they do indicate three relatively well-
2179 categorized conditions. They are all defined by the C Standard {2} and thus would
2180 have to be defined by any system with a C Standard {2} binding, even if not expli-
2181 citly included in POSIX.1.

There is very little that a Conforming POSIX.1 Application can do by catching, ignoring, or masking any of the signals SIGILL, SIGTRAP, SIGIOT, SIGEMT, SIGBUS, SIGSEGV, SIGSYS, or SIGFPE. They will generally be generated by the system only in cases of programming errors. While it may be desirable for some robust code (e.g., a library routine) to be able to detect and recover from programming errors in other code, these signals are not nearly sufficient for that purpose. One portable use that does exist for these signals is that a command interpreter can recognize them as the cause of a process's termination [with *wait()*] and print an appropriate message. The mnemonic tags for these signals are derived from their PDP-11 origin.

The signals SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU, and SIGCONT are provided for job control and are unchanged from 4.2BSD. The signal SIGCHLD is also typically used by job control shells to detect children that have terminated or, as in 4.2BSD, stopped. See also B.3.3.4.

Some implementations, including System V, have a signal named SIGCLD, which is similar to SIGCHLD in 4.2BSD. POSIX.1 permits implementations to have a single signal with both names. POSIX.1 carefully specifies ways in which portable applications can avoid the semantic differences between the two different implementations. The name SIGCHLD was chosen for POSIX.1 because most current application usages of it can remain unchanged in conforming applications. SIGCLD in System V has more cases of semantics that POSIX.1 does not specify, and thus applications using it are more likely to require changes in addition to the name change.

Some implementations that do not support job control may nonetheless implement SIGCHLD. Similarly, such an implementation may choose to implement SIGSTOP. Since POSIX.1 requires that symbolic names always be defined (with the exception of certain names in *<limits.h>* and *<unistd.h>*), a portable method of determining, at run-time, whether an optional signal is supported is to call the *sigaction()* function with *NULL act* and *oact* arguments. A successful return indicates that the signal is supported. Note that if *sysconf()* shows that job control is present, then all of the optional signals shall also be supported.

The signals SIGUSR1 and SIGUSR2 are commonly used by applications for notification of exceptional behavior and are described as “reserved as application defined” so that such use is not prohibited. Implementations should not generate SIGUSR1 or SIGUSR2, except when explicitly requested by *kill()*. It is recommended that libraries not use these two signals, as such use in libraries could interfere with their use by applications calling the libraries. If such use is unavoidable, it should be documented. It is prudent for nonportable libraries to use nonstandard signals to avoid conflicts with use of standard signals by portable libraries.

There is no portable way for an application to catch or ignore nonstandard signals. Some implementations define the range of signal numbers, so applications can install signal-catching functions for all of them. Unfortunately, implementation-defined signals often cause problems when caught or ignored by applications that do not understand the reason for the signal. While the desire exists for an application to be more robust by handling all possible signals [even those only generated by *kill()*], no existing mechanism was found to be sufficiently portable to include in POSIX.1. The value of such a mechanism, if included, would

2230 be diminished given that SIGKILL would still not be catchable.

2231 B.3.3.1.2 Signal Generation and Delivery

2232 The terms defined in this subclause are not used consistently in documentation of
2233 historical systems. Each signal can be considered to have a lifetime beginning
2234 with *generation* and ending with *delivery*. The POSIX.1 definition of *delivery* does
2235 not exclude ignored signals; this is considered a more consistent definition.

2236 Implementations should deliver unblocked signals as soon after they are gen-
2237 erated as possible. However, it is difficult for POSIX.1 to make specific require-
2238 ments about this, beyond those in *kill()* and *sigprocmask()*. Even on systems with
2239 prompt delivery, scheduling of higher priority processes is always likely to cause
2240 delays.

2241 In general, the interval between the generation and delivery of unblocked signals
2242 cannot be detected by an application. Thus, references to pending signals gen-
2243 erally apply to blocked, pending signals.

2244 In the 4.3BSD system, signals that are blocked and set to SIG_IGN are discarded
2245 immediately upon generation. For a signal that is ignored as its default action, if
2246 the action is SIG_DFL and the signal is blocked, a generated signal remains pend-
2247 ing. In the 4.1BSD system and in System V Release 3, two other implementations
2248 that support a somewhat similar signal mechanism, all ignored, blocked signals
2249 remain pending if generated. Because it is not normally useful for an application
2250 to simultaneously ignore and block the same signal, it was unnecessary for
2251 POSIX.1 to specify behavior that would invalidate any of the historical
2252 implementations.

2253 There is one case in some historical implementations where an unblocked, pend-
2254 ing signal does not remain pending until it is delivered. In the System V imple-
2255 mentation of *signal()*, pending signals are discarded when the action is set to
2256 SIG_DFL or a signal-catching routine (as well as to SIG_IGN). Except in the case
2257 of setting SIGCHLD to SIG_DFL, implementations that do this do not conform com-
2258 pletely to POSIX.1. Some earlier drafts of POSIX.1 explicitly stated this, but these
2259 statements were redundant due to the requirement that functions defined by
2260 POSIX.1 not change attributes of processes defined by POSIX.1 except as explicitly
2261 stated (see Section 3).

2262 POSIX.1 specifically states that the order in which multiple, simultaneously pend-
2263 ing signals are delivered is unspecified. This order has not been explicitly
2264 specified in historical implementations, but has remained quite consistent and
2265 been known to those familiar with the implementations. Thus, there have been
2266 cases where applications (usually system utilities) have been written with explicit
2267 or implicit dependencies on this order. Implementors and others porting existing
2268 applications may need to be aware of such dependencies.

2269 When there are multiple pending signals that are not blocked, implementations
2270 should arrange for the delivery of all signals at once, if possible. Some implemen-
2271 tations stack calls to all pending signal-catching routines, making it appear that
2272 each signal-catcher was interrupted by the next signal. In this case, the imple-
2273 mentation should ensure that this stacking of signals does not violate the seman-
2274 tics of the signal masks established by *sigaction()*. Other implementations pro-
2275 cess at most one signal when the operating system is entered, with remaining

signals saved for later delivery. Although this practice is widespread, this behavior is neither standardized nor endorsed. In either case, implementations should attempt to deliver signals associated with the current state of the process (e.g., SIGFPE) before other signals, if possible.

In 4.2BSD and 4.3BSD, it is not permissible to ignore or explicitly block SIGCONT because if blocking or ignoring this signal prevented it from continuing a stopped process, such a process could never be continued (only killed by SIGKILL). However, 4.2BSD and 4.3BSD do block SIGCONT during execution of its signal-catching function when it is caught, creating exactly this problem. A proposal was considered to disallow catching SIGCONT in addition to ignoring and blocking it, but this limitation led to objections. The consensus was to require that SIGCONT always continue a stopped process when generated. This removed the need to disallow ignoring or explicit blocking of the signal; note that SIG_IGN and SIG_DFL are equivalent for SIGCONT.

B.3.3.1.3 Signal Actions

Earlier drafts of POSIX.1 mentioned SIGCONT as a second exception to the rule that signals are not delivered to stopped processes until continued. Because POSIX.1 now specifies that SIGCONT causes the stopped process to continue when it is generated, delivery of SIGCONT is not prevented because a process is stopped, even without an explicit exception to this rule.

Ignoring a signal by setting the action to SIG_IGN (or SIG_DFL for signals whose default action is to ignore) is not the same as installing a signal-catching function that simply returns. Invoking such a function will interrupt certain system functions that block processes [e.g., *wait()*, *sigsuspend()*, *pause()*, *read()*, *write()*] while ignoring a signal has no such effect on the process.

Historical implementations discard pending signals when the action is set to SIG_IGN. However, they do not always do the same when the action is set to SIG_DFL and the default action is to ignore the signal. POSIX.1 requires this for the sake of consistency and also for completeness, since the only signal this applies to is SIGCHLD, and POSIX.1 disallows setting its action to SIG_IGN.

The specification of the effects of SIG_IGN on SIGCHLD as implementation defined permits, but does not require, the System V effect of causing terminating children to be ignored by *wait()*. Yet it permits SIGCHLD to be effectively ignored in an implementation-independent manner by use of SIG_DFL.

Some implementations (System V, for example) assign different semantics for SIGCHLD depending on whether the action is set to SIG_IGN or SIG_DFL. Since POSIX.1 requires that the default action for SIGCHLD be to ignore the signal, applications should always set the action to SIG_DFL in order to avoid SIGCHLD.

Some implementations (System V, for example) will deliver a SIGCHLD signal immediately when a process establishes a signal-catching function for SIGCHLD 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

2322 ensure that they have no children when altering the signal action for SIGCHLD
2323 thus need to be prepared for, but not depend on, generation of an immediate
2324 SIGCHLD signal.

2325 The default action of the stop signals (SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU) is
2326 to stop a process that is executing. If a stop signal is delivered to a process that is
2327 already stopped, it has no effect. In fact, if a stop signal is generated for a
2328 stopped process whose signal mask blocks the signal, the signal will never be
2329 delivered to the process since the process must receive a SIGCONT, which discards
2330 all pending stop signals, in order to continue executing.

2331 The SIGCONT signal shall continue a stopped process even if SIGCONT is blocked
2332 (or ignored). However, if a signal-catching routine has been established for
2333 SIGCONT, it will not be entered until SIGCONT is unblocked.

2334 If a process in an orphaned process group stops, it is no longer under the control
2335 of a job-control shell and hence would not normally ever be continued. Because of
2336 this, orphaned processes that receive terminal-related stop signals (SIGTSTP,
2337 SIGTTIN, SIGTTOU, but not SIGSTOP) must not be allowed to stop. The goal is to
2338 prevent stopped processes from languishing forever. [As SIGSTOP is sent only via
2339 *kill()*, it is assumed that the process or user sending a SIGSTOP can send a
2340 SIGCONT when desired.] Instead, the system must discard the stop signal. As an
2341 extension, it may also deliver another signal in its place. 4.3BSD sends a SIG-
2342 KILL, which is overly effective because SIGKILL is not catchable. Another possible
2343 choice is SIGHUP. 4.3BSD also does this for orphaned processes (processes whose
2344 parent has terminated) rather than for members of orphaned process groups; this
2345 is less desirable because job-control shells manage process groups. POSIX.1 also
2346 prevents SIGTTIN and SIGTTOU signals from being generated for processes in
2347 orphaned process groups as a direct result of activity on a terminal, preventing
2348 infinite loops when *read()* and *write()* calls generate signals that are discarded.
2349 (See B.7.1.1.4.) A similar restriction on the generation of SIGTSTP was con-
2350 sidered, but that would be unnecessary and more difficult to implement due to its
2351 asynchronous nature.

2352 Although POSIX.1 requires that signal-catching functions be called with only one
2353 argument, there is nothing to prevent conforming implementations from extend-
2354 ing POSIX.1 to pass additional arguments, as long as Strictly Conforming POSIX.1
2355 Applications continue to compile and execute correctly. Most historical implemen-
2356 tations do, in fact, pass additional, signal-specific arguments to certain signal-
2357 catching routines.

2358 There was a proposal to change the declared type of the signal handler to:

```
2359     void func (int sig, ... );
```

2360 The usage of ellipses ("*...*") is C Standard [2] syntax to indicate a variable
2361 number of arguments. Its use was intended to allow the implementation to pass
2362 additional information to the signal handler in a standard manner.

2363 Unfortunately, this construct would require all signal handlers to be defined with
2364 this syntax because the C Standard [2] allows implementations to use a different
2365 parameter passing mechanism for variable parameter lists than for nonvariable
2366 parameter lists. Thus, all existing signal handlers in all existing applications
2367 would have to be changed to use the variable syntax in order to be standard and

portable. This is in conflict with the goal of Minimal Changes to Existing Application Code.

When terminating a process from a signal-catching function, processes should be aware of any interpretation that their parent may make of the status returned by *wait()* or *waitpid()*. In particular, a signal-catching function should not call *exit(0)* or *_exit(0)* unless it wants to indicate successful termination. A nonzero argument to *exit()* or *_exit()* can be used to indicate unsuccessful termination. Alternatively, the process can use *kill()* to send itself a fatal signal (first ensuring that the signal is set to the default action and not blocked). (See also B.3.2.2).

The behavior of *unsafe* functions, as defined by this subclause, is undefined when they are invoked from signal-catching functions in certain circumstances. The behavior of reentrant functions, as defined by this subclause, is as specified by POSIX.1, regardless of invocation from a signal-catching function. This is the only intended meaning of the statement that reentrant functions may be used in signal-catching functions without restriction. Applications must still consider all effects of such functions on such things as data structures, files, and process state. In particular, application writers need to consider the restrictions on interactions when interrupting *sleep()* [see *sleep()* and B.3.4.3] and interactions among multiple handles for a file description (see 8.2.3 and B.8.2.3). The fact that any specific function is listed as reentrant does not necessarily mean that invocation of that function from a signal-catching function is recommended.

In order to prevent errors arising from interrupting nonreentrant function calls, applications should protect calls to these functions either by blocking the appropriate signals or through the use of some programmatic semaphore. POSIX.1 does not address the more general problem of synchronizing access to shared data structures. Note in particular that even the “safe” functions may modify the global variable *errno*; the signal-catching function may want to save and restore its value. The same principles apply to the reentrancy of application routines and asynchronous data access.

Note that *longjmp()* and *siglongjmp()* are not in the list of reentrant functions. This is because the code executing after *longjmp()* or *siglongjmp()* can call any unsafe functions with the same danger as calling those unsafe functions directly from the signal handler. Applications that use *longjmp()* or *siglongjmp()* out of signal handlers require rigorous protection in order to be portable. Many of the other functions that are excluded from the list are traditionally implemented using either the C language *malloc()* or *free()* functions or the C language standard I/O library, both of which traditionally use data structures in a nonreentrant manner. Because any combination of different functions using a common data structure can cause reentrancy problems, POSIX.1 does not define the behavior when any unsafe function is called in a signal handler that interrupts any unsafe function.

B.3.3.1.4 Signal Effects on Other Functions

The most common behavior of an interrupted function after a signal-catching function returns is for the interrupted function to give an [EINTR] error. However, there are a number of specific exceptions, including *sleep()* and certain situations with *read()* and *write()*.

2414 The historical implementations of many functions defined by POSIX.1 are not
 2415 interruptible, but delay delivery of signals generated during their execution until
 2416 after they complete. This is never a problem for functions that are guaranteed to
 2417 complete in a short (imperceptible to a human) period of time. It is normally
 2418 those functions that can suspend a process indefinitely or for long periods of time
 2419 [e.g., *wait()*, *pause()*, *sigsuspend()*, *sleep()*, or *read()/write()* on a slow device like a
 2420 terminal] that are interruptible. This permits applications to respond to interac-
 2421 tive signals or to set timeouts on calls to most such functions with *alarm()*.
 2422 Therefore, implementations should generally make such functions (including ones
 2423 defined as extensions) interruptible.

2424 Functions not mentioned explicitly as interruptible may be so on some implemen-
 2425 tations, possibly as an extension where the function gives an [EINTR] error.
 2426 There are several functions [e.g., *getpid()*, *getuid()*] that are specified as never
 2427 returning an error, which can thus never be extended in this way.

2428 B.3.3.2 Send a Signal to a Process

2429 The semantics for permission checking for *kill()* differ between System V and
 2430 most other implementations, such as Version 7 or 4.3BSD. The semantics chosen
 2431 for POSIX.1 agree with System V. Specifically, a set-user-ID process cannot pro-
 2432 tect itself against signals (or at least not against SIGKILL) unless it changes its
 2433 real user ID. This choice allows the user who starts an application to send it sig-
 2434 nals even if it changes its effective user ID. The other semantics give more power
 2435 to an application that wants to protect itself from the user who ran it.

2436 Some implementations provide semantic extensions to the *kill()* function when
 2437 the absolute value of *pid* is greater than some maximum, or otherwise special,
 2438 value. Negative values are a flag to *kill()*. Since most implementations return
 2439 [ESRCH] in this case, this behavior is not included in POSIX.1, although a con-
 2440 forming implementation could provide such an extension.

2441 The implementation-defined processes to which a signal cannot be sent may
 2442 include the scheduler or *init*.

2443 Most historical implementations use *kill(-1, sig)* from a super-user process to
 2444 send a signal to all processes (excluding system processes like *init*). This use of
 2445 the *kill()* function is for administrative purposes only; portable applications
 2446 should not send signals to processes about which they have no knowledge. In
 2447 addition, there are semantic variations among different implementations that,
 2448 because of the limited use of this feature, were not necessary to resolve by stan-
 2449 dardization. System V implementations also use *kill(-1, sig)* from a
 2450 nonsuper-user process to send a signal to all processes with matching user IDs.
 2451 This use was considered neither sufficiently widespread nor necessary for applica-
 2452 tion portability to warrant inclusion in POSIX.1.

2453 There was initially strong sentiment to specify that, if *pid* specifies that a signal
 2454 be sent to the calling process and that signal is not blocked, that signal would be
 2455 delivered before *kill()* returns. This would permit a process to call *kill()* and be
 2456 guaranteed that the call never return. However, historical implementations that
 2457 provide only the *signal()* interface make only the weaker guarantee in POSIX.1,
 2458 because they only deliver one signal each time a process enters the kernel.
 2459 Modifications to such implementations to support the *sigaction()* interface

generally require entry to the kernel following return from a signal-catching function, in order to restore the signal mask. Such modifications have the effect of satisfying the stronger requirement, at least when *sigaction()* is used, but not necessarily when *signal()* is used. The developers of POSIX.1 considered making the stronger requirement except when *signal()* is used, but felt this would be unnecessarily complex. Implementors are encouraged to meet the stronger requirement whenever possible. In practice, the weaker requirement is the same, except in the rare case when two signals arrive during a very short window. This reasoning also applies to a similar requirement for *sigprocmask()*.

In 4.2BSD, the SIGCONT signal can be sent to any descendant process regardless of user-ID security checks. This allows a job-control shell to continue a job even if processes in the job have altered their user IDs (as in the *su* command). In keeping with the addition of the concept of sessions, similar functionality is provided by allowing the SIGCONT signal to be sent to any process in the same session, regardless of user-ID security checks. This is less restrictive than BSD in the sense that ancestor processes (in the same session) can now be the recipient. It is more restrictive than BSD in the sense that descendant processes that form new sessions are now subject to the user-ID checks. A similar relaxation of security is not necessary for the other job-control signals since those signals are typically sent by the terminal driver in recognition of special characters being typed; the terminal driver bypasses all security checks.

In secure implementations, a process may be restricted from sending a signal to a process having a different security label. In order to prevent the existence or nonexistence of a process from being used as a covert channel, such processes should appear nonexistent to the sender; i.e., [ESRCH] should be returned, rather than [EPERM], if *pid* refers only to such processes.

Existing implementations vary on the result of a *kill()* with *pid* indicating an inactive process (a terminated process that has not been waited for by its parent). Some indicate success on such a call (subject to permission checking), while others give an error of [ESRCH]. Since POSIX.1's definition of *process lifetime* covers inactive processes, the [ESRCH] error as described is inappropriate in this case. In particular, this means that an application cannot have a parent process check for termination of a particular child with *kill()* [usually this is done with the null signal; this can be done reliably with *waitpid()*].

There is some belief that the name *kill()* is misleading, since the function is not always intended to cause process termination. However, the name is common to all historical implementations, and any change would be in conflict with the goal of Minimal Changes to Existing Application Code.

B.3.3.3 Manipulate Signal Sets

The implementation of the *sigemptyset()* [or *sigfillset()*] functions could quite trivially clear (or set) all the bits in the signal set. Alternatively, it would be reasonable to initialize part of the structure, such as a version field, to permit binary compatibility between releases where the size of the set varies. For such reasons, either *sigemptyset()* or *sigfillset()* must be called prior to any other use of the signal set, even if such use is read-only [e.g., as an argument to *sigpending()*]. This function is not intended for dynamic allocation.

2506 The *sigfillset()* and *sigemptyset()* functions require that the resulting signal set
2507 include (or exclude) all the signals defined in POSIX.1. Although it is outside the
2508 scope of POSIX.1 to place this requirement on signals that are implemented as
2509 extensions, it is recommended that implementation-defined signals also be
2510 affected by these functions. However, there may be a good reason for a particular
2511 signal not to be affected. For example, blocking or ignoring an implementation-
2512 defined signal may have undesirable side effects, whereas the default action for
2513 that signal is harmless. In such a case, it would be preferable for such a signal to
2514 be excluded from the signal set returned by *sigfillset()*.

2515 In earlier drafts of POSIX.1 there was no distinction between invalid and unsup-
2516 ported signals (the names of optional signals that were not supported by an
2517 implementation were not defined by that implementation). The [EINVAL] error
2518 was thus specified as a required error for invalid signals. With that distinction, it
2519 is not necessary to require implementations of these functions to determine
2520 whether an optional signal is actually supported, as that could have a significant
2521 performance impact for little value. The error could have been required for
2522 invalid signals and optional for unsupported signals, but this seemed unneces-
2523 sarily complex. Thus, the error is optional in both cases.

2524 B.3.3.4 Examine and Change Signal Action

2525 Although POSIX.1 requires that signals that cannot be ignored shall not be added
2526 to the signal mask when a signal-catching function is entered, there is no explicit
2527 requirement that subsequent calls to *sigaction()* reflect this in the information
2528 returned in the *oact* argument. In other words, if SIGKILL is included in the
2529 *sa_mask* field of *act*, it is unspecified whether or not a subsequent call to *sigac-*
2530 *tion()* will return with SIGKILL included in the *sa_mask* field of *oact*.

2531 The SA_NOCLDSTOP flag, when supplied in the *act->sa_flags* parameter, allows
2532 overloading SIGCHLD with the System V semantics that each SIGCLD signal indi-
2533 cates a single terminated child. Most portable applications that catch SIGCHLD
2534 are expected to install signal-catching functions that repeatedly call the *waitpid()*
2535 function with the WNOHANG flag set, acting on each child for which status is
2536 returned, until *waitpid()* returns zero. If stopped children are not of interest, the
2537 use of the SA_NOCLDSTOP flag can prevent the overhead from invoking the
2538 signal-catching routine when they stop.

2539 Some historical implementations also define other mechanisms for stopping
2540 processes, such as the *ptrace()* function. These implementations usually do not
2541 generate a SIGCHLD signal when processes stop due to this mechanism; however,
2542 that is beyond the scope of POSIX.1.

2543 POSIX.1 requires that calls to *sigaction()* that supply a NULL *act* argument
2544 succeed, even in the case of signals that cannot be caught or ignored (i.e., SIGKILL
2545 or SIGSTOP). The System V *signal()* and BSD *sigvec()* functions return [EINVAL]
2546 in these cases and, in this respect, their behavior varies from *sigaction()*.

2547 POSIX.1 requires that *sigaction()* properly save and restore a signal action set up
2548 by the C Standard {2} *signal()* function. However, there is no guarantee that the
2549 reverse is true, nor could there be given the greater amount of information con-
2550 veyed by the *sigaction* structure. Because of this, applications should avoid using
2551 both functions for the same signal in the same process. Since this cannot always

be avoided in case of general-purpose library routines, they should always be implemented with *sigaction()*.

It was intended that the *signal()* function should be implementable as a library routine using *sigaction()*.

B.3.3.5 Examine and Change Blocked Signals

When a process's signal mask is changed in a signal-catching function that is installed by *sigaction()*, the restoration of the signal mask on return from the signal-catching function overrides that change [see *sigaction()*]. If the signal-catching function was installed with *signal()*, it is unspecified whether this occurs.

See B.3.3.2 for a discussion of the requirement on delivery of signals.

B.3.3.6 Examine Pending Signals

There is no additional rationale provided for this subclause.

B.3.3.7 Wait for a Signal

Normally, at the beginning of a critical code section, a specified set of signals is blocked using the *sigprocmask()* function. When the process has completed the critical section and needs to wait for the previously blocked signal(s), it pauses by calling *sigsuspend()* with the mask that was returned by the *sigprocmask()* call.

B.3.4 Timer Operations

B.3.4.1 Schedule Alarm

Many historical implementations (including Version 7 and System V) allow an alarm to occur up to a second early. Other implementations allow alarms up to half a second or one clock tick early or do not allow them to occur early at all. The latter is considered most appropriate, since it gives the most predictable behavior, especially since the signal can always be delayed for an indefinite amount of time due to scheduling. Applications can thus choose the *seconds* argument as the minimum amount of time they wish to have elapse before the signal.

The term “real time” here and elsewhere [*sleep()*, *times()*] is intended to mean “wall clock” time as common English usage, and has nothing to do with “realtime operating systems.” It is in contrast to “virtual time,” which could be misinterpreted if just “time” were used.

In some implementations, including 4.3BSD, very large values of the *seconds* argument are silently rounded down to an implementation-defined maximum value. This maximum is large enough (on the order of several months) that the effect is not noticeable.

Application writers should note that the type of the argument *seconds* and the return value of *alarm()* is *unsigned int*. That means that a Strictly Conforming POSIX.1 Application cannot pass a value greater than the minimum guaranteed

2590 value for {UINT_MAX}, which the C Standard {2} sets as 65535, and any applica-
 2591 tion passing a larger value is restricting its portability. A different type was con-
 2592 sidered, but historical implementations, including those with a 16-bit *int* type,
 2593 consistently use either *unsigned int* or *int*.

2594 Application writers should be aware of possible interactions when the same pro-
 2595 cess uses both the *alarm()* and *sleep()* functions [see *sleep()* and B.3.4.3].

2596 B.3.4.2 Suspend Process Execution

2597 Many common uses of *pause()* have timing windows. The scenario involves check-
 2598 ing a condition related to a signal and, if the signal has not occurred, calling
 2599 *pause()*. When the signal occurs between the check and the call to *pause()*, the
 2600 process often blocks indefinitely. The *sigprocmask()* and *sigsuspend()* functions
 2601 can be used to avoid this type of problem.

2602 B.3.4.3 Delay Process Execution

2603 There are two general approaches to the implementation of the *sleep()* function.
 2604 One is to use the *alarm()* function to schedule a SIGALRM signal and then
 2605 suspend the process waiting for that signal. The other is to implement an
 2606 independent facility. POSIX.1 permits either approach.

2607 In order to comply with the wording of the introduction to Section 3, that no prim-
 2608 itive shall change a process attribute unless explicitly described by POSIX.1, an
 2609 implementation using SIGALRM must carefully take into account any SIGALRM
 2610 signal scheduled by previous *alarm()* calls, the action previously established for
 2611 SIGALRM, and whether SIGALRM was blocked. If a SIGALRM has been scheduled
 2612 before the *sleep()* would ordinarily complete, the *sleep()* must be shortened to that
 2613 time and a SIGALRM generated (possibly simulated by direct invocation of the
 2614 signal-catching function) before *sleep()* returns. If a SIGALRM has been scheduled
 2615 after the *sleep()* would ordinarily complete, it must be rescheduled for the same
 2616 time before *sleep()* returns. The action and blocking for SIGALRM must be saved
 2617 and restored.

2618 Historical implementations often implement the SIGALRM-based version using
 2619 *alarm()* and *pause()*. One such implementation is prone to infinite hangsups, as
 2620 described in B.3.4.2. Another such implementation uses the C language *setjmp()*
 2621 and *longjmp()* functions to avoid that window. That implementation introduces a
 2622 different problem: when the SIGALRM signal interrupts a signal-catching function
 2623 installed by the user to catch a different signal, the *longjmp()* aborts that signal-
 2624 catching function. An implementation based on *sigprocmask()*, *alarm()*, and *sig-*
 2625 *suspend()* can avoid these problems.

2626 Despite all reasonable care, there are several very subtle, but detectable and una-
 2627 voidable, differences between the two types of implementations. These are the
 2628 cases mentioned in POSIX.1 where some other activity relating to SIGALRM takes
 2629 place, and the results are stated to be unspecified. All of these cases are
 2630 sufficiently unusual as not to be of concern to most applications.

2631 (See also the discussion of the term “real time” in B.3.4.1.)

Because *sleep()* can be implemented using *alarm()*, the discussion about alarms occurring early under B.3.4.1 applies to *sleep()* as well.

Application writers should note that the type of the argument *seconds* and the return value of *sleep()* is *unsigned int*. That means that a Strictly Conforming POSIX.1 Application cannot pass a value greater than the minimum guaranteed value for `UINT_MAX`, which the C Standard {2} sets as 65535, and any application passing a larger value is restricting its portability. A different type was considered, but historical implementations, including those with a 16-bit *int* type, consistently use either *unsigned int* or *int*.

Scheduling delays may cause the process to return from the *sleep()* function significantly after the requested time. In such cases, the return value should be set to zero, since the formula (requested time minus the time actually spent) yields a negative number and *sleep()* returns an *unsigned int*.

B.4 Process Environment

B.4.1 Process Identification

B.4.1.1 Get Process and Parent Process IDs

There is no additional rationale provided for this subclause.

B.4.2 User Identification

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

There is no additional rationale provided for this subclause.

B.4.2.2 Set User and Group IDs

The saved set-user-ID capability allows a program to regain the effective user ID established at the last *exec* call. Similarly, the saved set-group-ID capability allows a program to regain the effective group ID established at the last *exec* call.

These two capabilities are derived from System V. Without them, a program may have to run as super-user in order to perform the same functions, because super-user can write on the user's files. This is a problem because such a program can write on *any* user's files, and so must be carefully written to emulate the permissions of the calling process properly.

A process with appropriate privilege on a system with this saved ID capability establishes all relevant IDs to the new value, since this function is used to establish the identity of the user during *login* or *su*. Any change to this behavior would be dangerous since it involves programs that need to be trusted.

2666 The behavior of 4.2BSD and 4.3BSD that allows setting the real ID to the effective
2667 ID is viewed as a value-dependent special case of appropriate privilege.

2668 B.4.2.3 Get Supplementary Group IDs

2669 The related function *setgroups()* is a privileged operation and therefore is not
2670 covered by POSIX.1.

2671 As implied by the definition of supplementary groups, the effective group ID may
2672 appear in the array returned by *getgroups()* or it may be returned only by
2673 *getegid()*. Duplication may exist, but the application needs to call *getegid()* to be
2674 sure of getting all of the information. Various implementation variations and
2675 administrative sequences will cause the set of groups appearing in the result of
2676 *getgroups()* to vary in order and as to whether the effective group ID is included,
2677 even when the set of groups is the same (in the mathematical sense of “set”). (The
2678 history of a process and its parents could affect the details of result.)

2679 Applications writers should note that {NGROUPS_MAX} is not necessarily a con-
2680 stant on all implementations.

2681 B.4.2.4 Get User Name

2682 The *getlogin()* function returns a pointer to the user’s login name. The same user
2683 ID may be shared by several login names. If it is desired to get the user database
2684 entry that is used during login, the result of *getlogin()* should be used to provide
2685 the argument to the *getpwnam()* function. (This might be used to determine the
2686 user’s login shell, particularly where a single user has multiple login shells with
2687 distinct login names, but the same user ID.)

2688 The information provided by the *cuserid()* function, which was originally defined
2689 in IEEE Std 1003.1-1990 and subsequently removed, can be obtained by the
2690 following:

2691 getpwuid(geteuid())

2692 while the information provided by historical implementations of *cuserid()* can be
2693 obtained by:

2694 getpwuid(getuid())

2695 B.4.3 Process Groups

2696 B.4.3.1 Get Process Group ID

2697 4.3BSD provides a *getpgrp()* function that returns the process group ID for a
2698 specified process. Although this function is used to support job control, all known
2699 job-control shells always specify the calling process with this function. Thus, the
2700 simpler System V *getpgrp()* suffices, and the added complexity of the 4.3BSD
2701 *getpgrp()* has been omitted from POSIX.1.

2 B.4.3.2 Create Session and Set Process Group ID

3 The *setsid()* function is similar to the *setpgrp()* function of System V. System V,
4 without job control, groups processes into process groups and creates new process
5 groups via *setpgrp()*; only one process group may be part of a login session.

6 Job control allows multiple process groups within a login session. In order to
7 limit job-control actions so that they can only affect processes in the same login
8 session, POSIX.1 adds the concept of a session that is created via *setsid()*. The *set-*
9 *sid()* function also creates the initial process group contained in the session.
0 Additional process groups can be created via the *setpgid()* function. A System V
1 process group would correspond to a POSIX.1 session containing a single POSIX.1
2 process group. Note that this function requires that the calling process not be a
3 process group leader. The usual way to ensure this is true is to create a new pro-
4 cess with *fork()* and have it call *setsid()*. The *fork()* function guarantees that the
5 process ID of the new process does not match any existing process group ID.

6 B.4.3.3 Set Process Group ID for Job Control

7 The *setpgid()* function is used to group processes together for the purpose of sig-
8 naling, placement in foreground or background, and other job-control actions. See
9 B.2.2.2.

0 The *setpgid()* function is similar to the *setpgrp()* function of 4.2BSD, except that
1 4.2BSD allowed the specified new process group to assume any value. This
2 presents certain security problems and is more flexible than necessary to support
3 job control.

4 To provide tighter security, *setpgid()* only allows the calling process to join a pro-
5 cess group already in use inside its session or create a new process group whose
6 process group ID was equal to its process ID.

7 When a job-control shell spawns a new job, the processes in the job must be
8 placed into a new process group via *setpgid()*. There are two timing constraints
9 involved in this action:

- 0 (1) The new process must be placed in the new process group before the
1 appropriate program is launched via one of the *exec* functions.
- 2 (2) The new process must be placed in the new process group before the shell
3 can correctly send signals to the new process group.

4 To address these constraints, the following actions are performed: The new
5 processes call *setpgid()* to alter their own process groups after *fork()* but before
6 *exec*. This satisfies the first constraint. Under 4.3BSD, the second constraint is
7 satisfied by the synchronization property of *vfork()*; that is, the shell is suspended
8 until the child has completed the *exec*, thus ensuring that the child has completed
9 the *setpgid()*. A new version of *fork()* with this same synchronization property
0 was considered, but it was decided instead to merely allow the parent shell pro-
1 cess to adjust the process group of its child processes via *setpgid()*. Both timing
2 constraints are now satisfied by having both the parent shell and the child
3 attempt to adjust the process group of the child process; it does not matter which
4 succeeds first.

2745 Because it would be confusing to an application to have its process group change
 2746 after it began executing (i.e., after *exec*) and because the child process would
 2747 already have adjusted its process group before this, the [EACCES] error was added
 2748 to disallow this.

2749 One nonobvious use of *setpgid()* is to allow a job-control shell to return itself to its
 2750 original process group (the one in effect when the job-control shell was executed).
 2751 A job-control shell does this before returning control back to its parent when it is
 2752 terminating or suspending itself as a way of restoring its job control “state” back
 2753 to what its parent would expect. (Note that the original process group of the job-
 2754 control shell typically matches the process group of its parent, but this is not
 2755 necessarily always the case.) See also B.7.2.4.

2756 B.4.4 System Identification

2757 B.4.4.1 System Name

2758 The values of the structure members are not constrained to have any relation to
 2759 the version of POSIX.1 implemented in the operating system. An application
 2760 should instead depend on `{_POSIX_VERSION}` and related constants defined in 2.9.

2761 POSIX.1 does not define the sizes of the members of the structure and permits
 2762 them to be of different sizes, although most implementations define them all to be
 2763 the same size: eight bytes plus one byte for the string terminator. That size for
 2764 *nodename* is not enough for use with many networks.

2765 The *uname()* function is specific to System III, System V, and related implementa-
 2766 tions, and it does not exist in Version 7 or 4.3BSD. The values it returns are set
 2767 at system compile time in those historical implementations.

2768 4.3BSD has *gethostname()* and *gethostid()*, which return a symbolic name and a
 2769 numeric value, respectively. There are related *sethostname()* and *sethostid()*
 2770 functions that are used to set the values the other two functions return. The
 2771 length of the host name is limited to 31 characters in most implementations and
 2772 the host ID is a 32-bit integer.

2773 B.4.5 Time

2774 The *time()* function returns a value in seconds (type *time_t*) while *times()* returns
 2775 a set of values in clock ticks (type *clock_t*). Some historical implementations, such
 2776 as 4.3BSD, have mechanisms capable of returning more precise times [see the
 2777 description of *gettimeofday()* in B.4.5.1]. A generalized timing scheme to unify
 2778 these various timing mechanisms has been proposed but not adopted in POSIX.1.

2779 B.4.5.1 Get System Time

2780 Implementations in which *time_t* is a 32-bit signed integer (most historical imple-
 2781 mentations) will fail in the year 2038. This version of POSIX.1 does not address
 2782 this problem. However, the use of the new *time_t* type is mandated in order to
 2783 ease the eventual fix.

The use of the header `<time.h>`, instead of `<sys/types.h>`, allows compatibility with the C Standard [2].

Many historical implementations (including Version 7) and the 1984 */usr/group Standard* [B59] use *long* instead of *time_t*. POSIX.1 uses the latter type in order to agree with the C Standard [2].

4.3BSD includes *time()* only as an interface to the more flexible *gettimeofday()* function.

B.4.5.2 Get Process Times

The accuracy of the times reported is intentionally left unspecified to allow implementations flexibility in design, from uniprocessor to multiprocessor networks.

The inclusion of times of child processes is recursive, so that a parent process may collect the total times of all of its descendants. But the times of a child are only added to those of its parent when its parent successfully waits on the child. Thus, it is not guaranteed that a parent process will always be able to see the total times of all its descendants.

(See also the discussion of the term “real time” in B.3.4.1.)

If the type *clock_t* is defined to be a signed 32-bit integer, it will overflow in somewhat more than a year if there are 60 clock ticks per second, or less than a year if there are 100. There are individual systems that run continuously for longer than that. POSIX.1 permits an implementation to make the reference point for the returned value be the startup time of the process, rather than system startup time.

The term “charge” in this context has nothing to do with billing for services. The operating system accounts for time used in this way. That information must be correct, regardless of how that information is used.

B.4.6 Environment Variables

B.4.6.1 Environment Access

Additional functions *putenv()* and *clearenv()* were considered but rejected because they were considered to be more oriented towards system administration than ordinary application programs. This is being reconsidered for an amendment to POSIX.1 because uses from within an application have been identified since the decision was made.

It was proposed that this function is properly part of Section 8. It is an extension to a function in the C Standard [2]. Because this function should be available from any language, not just C, it appears here, to separate it from the material in Section 8, which is specific to the C binding. (The localization extensions to C are not, at this time, appropriate for other languages.)

2821 B.4.7 Terminal Identification

2822 The difference between *ctermid()* and *ttyname()* is that *ttyname()* must be passed
 2823 a file descriptor and returns the pathname of the terminal associated with that
 2824 file descriptor, while *ctermid()* returns a string (such as */dev/tty*) that will refer
 2825 to the controlling terminal if used as a pathname. Thus *ttyname()* is useful only if
 2826 the process already has at least one file open to a terminal.

2827 The historical value of *ctermid()* is */dev/tty*; this is acceptable. The *ctermid()*
 2828 function should not be used to determine if a process actually has a controlling
 2829 terminal, but merely the name that would be used.

2830 B.4.7.1 Generate Terminal Pathname

2831 *L_ctermid* must be defined appropriately for a given implementation and must be
 2832 greater than zero so that array declarations using it are accepted by the compiler.
 2833 The value includes the terminating null byte.

2834 B.4.7.2 Determine Terminal Device Name

2835 The term “terminal” is used instead of the historical term “terminal device” in
 2836 order to avoid a reference to an undefined term.

2837 B.4.8 Configurable System Variables

2838 This subclause was added in response to requirements of application developers |
 2839 and of system vendors who deal with many international system configurations. |
 2840 It is closely related to B.5.7 as well.

2841 Although a portable application can run on all systems by never demanding more
 2842 resources than the minimum values published in POSIX.1, it is useful for that
 2843 application to be able to use the actual value for the quantity of a resource avail-
 2844 able on any given system. To do this, the application will make use of the value of
 2845 a symbolic constant in *<limits.h>* or *<unistd.h>*.

2846 However, once compiled, the application must still be able to cope if the amount of
 2847 resource available is increased. To that end, an application may need a means of
 2848 determining the quantity of a resource, or the presence of an option, at execution
 2849 time.

2850 Two examples are offered:

- 2851 (1) Applications may wish to act differently on systems with or without job
 2852 control. Applications vendors who wish to distribute only a single binary
 2853 package to all instances of a computer architecture would be forced to
 2854 assume job control is never available if it were to rely solely on the
 2855 *<unistd.h>* value published in POSIX.1.
- 2856 (2) International applications vendors occasionally require knowledge of the |
 2857 number of clock ticks per second. Without the facilities of this subclause, |
 2858 they would be required to either distribute their applications partially in |
 2859 source form or to have 50 Hz and 60 Hz versions for the various countries |
 2860 in which they operate.

1 It is the knowledge that many applications are actually distributed widely in exe-
2 cutable form that lead to this facility. If limited to the most restrictive values in
3 the headers, such applications would have to be prepared to accept the most lim-
4 ited environments offered by the smallest microcomputers. Although this is
5 entirely portable, there was a consensus that they should be able to take advan-
6 tage of the facilities offered by large systems, without the restrictions associated
7 with source and object distributions.

8 During the discussions of this feature, it was pointed out that it is almost always
9 possible for an application to discern what a value might be at run-time by suit-
0 ably testing the various interfaces themselves. And, in any event, it could always
1 be written to adequately deal with error returns from the various functions. In
2 the end, it was felt that this imposed an unreasonable level of complication and
3 sophistication on the application writer.

4 This run-time facility is not meant to provide ever-changing values that applica-
5 tions will have to check multiple times. The values are seen as changing no more
6 frequently than once per system initialization, such as by a system administrator
7 or operator with an automatic configuration program. POSIX.1 specifies that they
8 shall not change within the lifetime of the process.

9 Some values apply to the system overall and others vary at the file system or
10 directory level. These latter are described in B.5.7.

11 **B.4.8.1 Get Configurable System Variables**

12 Note that all values returned must be expressible as integers. String values were
13 considered, but the additional flexibility of this approach was rejected due to its
14 added complexity of implementation and use.

15 Some values, such as {PATH_MAX}, are sometimes so large that they must not be
16 used to, say, allocate arrays. The *sysconf()* function will return a negative value
17 to show that this symbolic constant is not even defined in this case.

18 **B.4.8.1.1 Special Symbol {CLK_TCK}**

19 {CLK_TCK} appears in POSIX.1 for backwards compatibility with IEEE Std
20 1003.1-1988. Its use is obsolescent.

21 **B.4.8.2 Get Password From User**

22 The *getpass()* function was explicitly excluded from POSIX.1 because it was found
23 that the name was misleading, and it provided no functionality that the user
24 could not easily implement within POSIX.1. The implication of some form of secu-
25 rity, which was not actually provided, exceeded the small gain in convenience.

2896 **B.5 Files and Directories**2897 See *pathname resolution*.

2898 The wording regarding the group of a newly created regular file, directory, or
 2899 FIFO in *open()*, *mkdir()*, *mkfifo()*, respectively, defines the two acceptable
 2900 behaviors in order to permit both the System V (and Version 7) behavior (in which
 2901 the group of the new object is set to the effective group ID of the creating process)
 2902 and the 4.3BSD behavior (in which the new object has the group of its parent
 2903 directory). An application that needs a file to be created specifically in one or the
 2904 other of the possible groups should use *chown()* to ensure the new group regard-
 2905 less of the style of groups the interface implements. Most applications will not
 2906 and should not be concerned with the group ID of the file.

2907 **B.5.1 Directories**

2908 Historical implementations prior to 4.2BSD had no special functions, types, or
 2909 headers for directory access. Instead, directories were read with *read()* and each
 2910 program that did so had code to understand the internal format of directory files.
 2911 Many such programs did not correctly handle the case of a maximum-length (his-
 2912 torically fourteen character) filename and would neglect to add a null character
 2913 string terminator when doing comparisons. The access methods in POSIX.1 elim-
 2914 inate that bug, as well as hiding differences in implementations of directories or
 2915 file systems.

2916 The directory access functions originally selected for POSIX.1 were derived from
 2917 4.2BSD, were adopted in System V Release 3, and are in *SVID* [B39] Volume 3,
 2918 with the exception of a type difference for the *d_ino* field. That field represents
 2919 implementation-dependent or even file system-dependent information (the i-node
 2920 number in most implementations). Since the directory access mechanism is
 2921 intended to be implementation-independent, and since only system programs, not
 2922 ordinary applications, need to know about the i-node number (or file serial
 2923 number) in this context, the *d_ino* field does not appear in POSIX.1. Also, pro-
 2924 grams that want this information can get it with *stat()*.

2925 **B.5.1.1 Format of Directory Entries**

2926 Information similar to that in the header `<dirent.h>` is contained in a file
 2927 `<sys/dir.h>` in 4.2BSD and 4.3BSD. The equivalent in these implementations
 2928 of *struct dirent* from POSIX.1 is *struct direct*. The filename was changed because
 2929 the name `<sys/dir.h>` was also used in earlier implementations to refer to
 2930 definitions related to the older access method; this produced name conflicts. The
 2931 name of the structure was changed because POSIX.1 does not completely define
 2932 what is in the structure, so it could be different on some implementations from
 2933 *struct direct*.

2934 The name of an array of *char* of an unspecified size should not be used as an
 2935 *lvalue*. Use of

2936 `sizeof (d_name)`

2937 is incorrect; use

38 `strlen (d_name)`

39 *instead.*

40 The array of *char d_name* is not a fixed size. Implementations may need to
41 declare *struct dirent* with an array size for *d_name* of 1, but the actual number of
42 characters provided matches (or only slightly exceeds) the length of the file name.

43 Currently, implementations are excluded if they have *d_name* with type *char **.
44 Lacking experience of such implementations, the developers of POSIX.1 declined
45 to try to describe in standards language what to do if either type were permitted.

46 **B.5.1.2 Directory Operations**

47 Based on historical implementations, the rules about file descriptors apply to
48 directory streams as well. However, POSIX.1 does not mandate that the directory
49 stream be implemented using file descriptors. The description of *opendir()*
50 clarifies that if a file descriptor is used for the directory stream it is mandatory
51 that *closedir()* deallocate the file descriptor. When a file descriptor is used to
52 implement the directory stream, it behaves as if the *FD_CLOEXEC* had been set
53 for the file descriptor.

54 The returned value of *readdir()* merely *represents* a directory entry. No
55 equivalence should be inferred.

56 The directory entries for dot and dot-dot are optional. POSIX.1 does not provide a
57 way to test *a priori* for their existence because an application that is portable
58 must be written to look for (and usually ignore) those entries. Writing code that
59 presumes that they are the first two entries does not always work, as many imple-
60 mentations permit them to be other than the first two entries, with a “normal”
61 entry preceding them. There is negligible value in providing a way to determine
62 what the implementation does because the code to deal with dot and dot-dot must
63 be written in any case and because such a flag would add to the list of those flags
64 (which has proven in itself to be objectionable) and might be abused.

55 Since the structure and buffer allocation, if any, for directory operations are
56 defined by the implementation, POSIX.1 imposes no portability requirements for
57 erroneous program constructs, erroneous data, or the use of indeterminate values
58 such as the use or referencing of a *dirp* value or a *dirent* structure value after a
59 directory stream has been closed or after a *fork()* or one of the *exec* function calls.

70 Historical implementations of *readdir()* obtain multiple directory entries on a sin-
71 gle read operation, which permits subsequent *readdir()* operations to operate
72 from the buffered information. Any wording that required each successful *read-*
73 *dir()* operation to mark the directory *st_atime* field for update would militate
74 against the historical performance-oriented implementations.

75 Since *readdir()* returns NULL both:

- 76 (1) When it detects an error, and
- 77 (2) When the end of the directory is encountered

78 an application that needs to tell the difference must set *errno* to zero before the
79 call and check it if NULL is returned. Because the function must not change
80 *errno* in case (2) and must set it to a nonzero value in case (1), a zero *errno* after a

2981 call returning NULL indicates end of directory, otherwise an error.

2982 Routines to deal with this problem more directly were proposed:

```
2983     int derror (dirp)
2984     DIR *dirp;

2985     void clearerr (dirp)
2986     DIR *dirp;
```

2987 The first would indicate whether an error had occurred, and the second would
2988 clear the error indication. The simpler method involving *errno* was adopted
2989 instead by requiring that *readdir()* not change *errno* when end-of-directory is
2990 encountered.

2991 Historical implementations include two more functions:

```
2992     long telldir (dirp)
2993     DIR *dirp;

2994     void seekdir (dirp, loc)
2995     DIR *dirp;
2996     long loc;
```

2997 The *telldir()* function returns the current location associated with the named
2998 directory stream.

2999 The *seekdir()* function sets the position of the next *readdir()* operation on the
3000 directory stream. The new position reverts to the one associated with the direc-
3001 tory stream when the *telldir()* operation was performed.

3002 These functions have restrictions on their use related to implementation details.
3003 Their capability can usually be accomplished by saving a filename found by *read-*
3004 *dir()* and later using *rewinddir()* and a loop on *readdir()* to relocate the position
3005 from which the filename was saved. Though this method is probably slower than
3006 using *seekdir()* and *telldir()*, there are few applications in which the capability is
3007 needed. Furthermore, directory systems that are implemented using technology
3008 such as balanced trees, where the order of presentation may vary from access to
3009 access, do not lend themselves well to any concept along these lines. For these
3010 reasons, *seekdir()* and *telldir()* are not included in POSIX.1.

3011 An error or signal indicating that a directory has changed while open was con-
3012 sidered but rejected.

3013 B.5.1.3 Set Position of Directory Stream

3014 The *seekdir()* and *telldir()* functions were proposed for inclusion in POSIX.1, but
3015 were excluded because they are inherently unreliable when all the possible con-
3016 forming implementations of the rest of POSIX.1 were considered. The problem is
3017 that returning to a given point in a directory is quite difficult to describe formally,
3018 in spite of its intuitive appeal, when systems that used B-trees, hashing functions,
3019 or other similar mechanisms for directory search are considered.

3020 Even the simple goal of attempting to visit each directory entry that is unmodified
3021 between the *opendir()* and *closedir()* calls exactly once is difficult to implement
3022 reliably in the face of directory compaction and reorganization.

Since the primary need for *seekdir()* and *telldir()* is to implement file tree walks, and since such a function is likely to be included in a future revision of POSIX.1, and since in that more constrained context it appears that at least the goal of visiting unmodified nodes exactly once can be achieved, it was felt that waiting for the development of that function best served all the constituencies.

B.5.2 Working Directory

B.5.2.1 Change Current Working Directory

The *chdir()* function only affects the working directory of the current process.

The result if a NULL argument is passed to *chdir()* is left implementation defined because some implementations dynamically allocate space in that case.

B.5.2.2 Working Directory Pathname

Since the maximum pathname length is arbitrary unless {PATH_MAX} is defined, an application generally cannot supply a *buf* with size {(PATH_MAX) + 1}.

Having *getcwd()* take no arguments and instead use the C function *malloc()* to produce space for the returned argument was considered. The advantage is that *getcwd()* knows how big the working directory pathname is and can allocate an appropriate amount of space. But the programmer would have to use the C function *free()* to free the resulting object, or each use of *getcwd()* would further reduce the available memory. Also, *malloc()* and *free()* are used nowhere else in POSIX.1. Finally, *getcwd()* is taken from the SVID [B39], where it has the two arguments used in POSIX.1.

The older function *getwd()* was rejected for use in this context because it had only a buffer argument and no size argument, and thus had no way to prevent overwriting the buffer, except to depend on the programmer to provide a large enough buffer.

The result if a NULL argument is passed to *getcwd()* is left implementation defined because some implementations dynamically allocate space in that case.

If a program is operating in a directory where some (grand)parent directory does not permit reading, *getcwd()* may fail, as in most implementations it must read the directory to determine the name of the file. This can occur if search, but not read, permission is granted in an intermediate directory, or if the program is placed in that directory by some more privileged process (e.g., *login*). Including this error, [EACCES], makes the reporting of the error consistent and warns the application writer that *getcwd()* can fail for reasons beyond the control of the application writer or user. Some implementations can avoid this occurrence [e.g., by implementing *getcwd()* using *pwd*, where *pwd* is a set-user-root process], thus the error was made optional.

Because POSIX.1 permits the addition of other errors, this would be a common addition and yet one that applications could not be expected to deal with without this addition.

3063 Some current implementations use (PATH_MAX)+2 bytes. These will have to be
 3064 changed. Many of those same implementations also may not diagnose the
 3065 [ERANGE] error properly or deal with a common bug having to do with newline in
 3066 a directory name (the fix to which is essentially the same as the fix for using +1
 3067 bytes), so this is not a severe hardship.

3068 B.5.2.3 Change Process's Root Directory

3069 The *chroot()* function was excluded from POSIX.1 on the basis that it was not use-
 3070 ful to portable applications. In particular, creating an environment in which an
 3071 application could run after executing a *chroot()* call is well beyond the current
 3072 scope of POSIX.1.

3073 B.5.3 General File Creation

3074 Because there is no portable way to specify a value for the argument indicating
 3075 the file mode bits (except zero), `<sys/stat.h>` is included with the functions
 3076 that reference mode bits.

3077 B.5.3.1 Open a File

3078 Except as specified in POSIX.1, the flags allowed in *oflag* are not mutually
 3079 exclusive and any number of them may be used simultaneously.

3080 Some implementations permit opening FIFOs with O_RDWR. Since FIFOs could
 3081 be implemented in other ways, and since two file descriptors can be used to the
 3082 same effect, this possibility is left as undefined.

3083 See B.4.2.3 about the group of a newly created file.

3084 The use of *open()* to create a regular file is preferable to the use of *creat()* because
 3085 the latter is redundant and included only for historical reasons.

3086 The use of the O_TRUNC flag on FIFOs and directories [pipes cannot be *open()*-ed]
 3087 must be permissible without unexpected side effects [e.g., *creat()* on a FIFO must
 3088 not remove data]. Because terminal special files might have type-ahead data
 3089 stored in the buffer, O_TRUNC should not affect their content, particularly if a
 3090 program that normally opens a regular file should open the current controlling
 3091 terminal instead. Other file types, particularly implementation-defined ones, are
 3092 left implementation defined.

3093 Implementations may deny access and return [EACCES] for reasons other than
 3094 just those listed in the [EACCES] definition.

3095 The O_NOCTTY flag was added to allow applications to avoid unintentionally
 3096 acquiring a controlling terminal as a side effect of opening a terminal file.
 3097 POSIX.1 does not specify how a controlling terminal is acquired, but it allows an
 3098 implementation to provide this on *open()* if the O_NOCTTY flag is not set and
 3099 other conditions specified in 7.1.1.3 are met. The O_NOCTTY flag is an effective
 3100 no-op if the file being opened is not a terminal device.

3101 In historical implementations the value of O_RDONLY is zero. Because of that, it
 3102 is not possible to detect the presence of O_RDONLY and another option. Future

implementations should encode `O_RDONLY` and `O_WRONLY` as bit flags so that:

`O_RDONLY | O_WRONLY == O_RDWR`

See the rationale for the change from `O_NDELAY` to `O_NONBLOCK` in B.6.

B.5.3.2 Create a New File or Rewrite an Existing One

The *creat()* function is redundant. Its services are also provided by the *open()* function. It has been included primarily for historical purposes since many existing applications depend on it. It is best considered a part of the C binding rather than a function that should be provided in other languages.

B.5.3.3 Set File Creation Mask

Unsigned argument and return types for *umask()* were proposed. The return type and the argument were both changed to *mode_t*.

Historical implementations have made use of additional bits in *cmask* for their implementation-specific purposes. The addition of the text that the meaning of other bits of the field are implementation defined permits these implementations to conform to POSIX.1.

B.5.3.4 Link to a File

See B.2.2.2.

Linking to a directory is restricted to the super-user in most historical implementations because this capability may produce loops in the file hierarchy or otherwise corrupt the file system. POSIX.1 continues that philosophy by prohibiting *link()* and *unlink()* from doing this. Other functions could do it if the implementor designed such an extension.

Some historical implementations allow linking of files on different file systems. Wording was added to explicitly allow this optional behavior. Symbolic links are not discussed by POSIX.1. The exception for cross-file system links is intended to apply only to links that are programmatically indistinguishable from “hard” links.

B.5.4 Special File Creation

B.5.4.1 Make a Directory

See B.2.5.

The *mkdir()* function originated in 4.2BSD and was added to System V in Release 3.0.

4.3BSD detects [ENAMETOOLONG].

See B.4.2.3 about the group of a newly created directory.

3136 **B.5.4.2 Make a FIFO Special File**

3137 The syntax of this routine is intended to maintain compatibility with historical
 3138 implementations of *mknod()*. The latter function was included in the 1984
 3139 */usr/group Standard* (B59), but only for use in creating FIFO special files. The
 3140 *mknod()* function was excluded from POSIX.1 as implementation defined and
 3141 replaced by *mkdir()* and *mkfifo()*.

3142 See B.4.2.3 about the group of a newly created FIFO.

3143 **B.5.5 File Removal**

3144 The *rmdir()* and *rename()* functions originated in 4.2BSD, and they used
 3145 [ENOTEMPTY] for the condition when the directory to be removed does not exist
 3146 or *new* already exists. When the 1984 */usr/group Standard* (B59) was published,
 3147 it contained [EEXIST] instead. When these functions were adopted into System V,
 3148 the 1984 */usr/group Standard* (B59) was used as a reference. Therefore, several
 3149 existing applications and implementations support/use both forms, and no agree-
 3150 ment could be reached on either value. All implementations are required to sup-
 3151 ply both [EEXIST] and [ENOTEMPTY] in *<errno.h>* with distinct values so that
 3152 applications can use both values in C language case statements.

3153 **B.5.5.1 Remove Directory Entries**

3154 Unlinking a directory is restricted to the super-user in many historical implemen-
 3155 tations for reasons given in B.5.3.4. But see B.5.5.3.

3156 The meaning of [EBUSY] in historical implementations is “mount point busy.”
 3157 Since POSIX.1 does not cover the system administration concepts of mounting and
 3158 unmounting, the description of the error was changed to “resource busy.” (This
 3159 meaning is used by some device drivers when a second process tries to open an
 3160 exclusive use device.) The wording is also intended to allow implementations to
 3161 refuse to remove a directory if it is the root or current working directory of any
 3162 process.

3163 **B.5.5.2 Remove a Directory**

3164 See also B.5.5 and B.5.5.1.

3165 **B.5.5.3 Rename a File**

3166 This *rename()* function is equivalent for regular files to that defined by the
 3167 C Standard [2]. Its inclusion here expands that definition to include actions on
 3168 directories and specifies behavior when the *new* parameter names a file that
 3169 already exists. That specification requires that the action of the function be
 3170 atomic.

3171 One of the reasons for introducing this function was to have a means of renaming
 3172 directories while permitting implementations to prohibit the use of *link()* and
 3173 *unlink()* with directories, thus constraining links to directories to those made by
 3174 *mkdir()*.

The specification that if *old* and *new* refer to the same file describes existing, although undocumented, 4.3BSD behavior. It is intended to guarantee that:

```
rename("x", "x");
```

does not remove the file.

Renaming *dot* or *dot-dot* is prohibited in order to prevent cyclical file system paths.

See also the descriptions of [ENOTEMPTY] and [ENAMETOOLONG] in B.5.5 and [EBUSY] in B.5.5.1. For a discussion of [EXDEV], see B.5.3.4.

B.5.6 File Characteristics

The *ustat()* function, which appeared in the 1984 */usr/group Standard* (B59) and is still in the *SVID* (B39), was excluded from POSIX.1 because it is:

- Not reliable. The amount of space available can change between the time the call is made and the time the calling process attempts to use it.
- Not required. The only known program that uses it is the text editor *ed*.
- Not readily extensible to networked systems.

B.5.6.1 File Characteristics: Header and Data Structure

See B.2.5.

A conforming C language application must include `<sys/stat.h>` for functions that have arguments or return values of type *mode_t*, so that symbolic values for that type can be used. An alternative would be to require that these constants are also defined by including `<sys/types.h>`.

The *S_ISUID* and *S_ISGID* bits may be cleared on any write, not just on *open()*, as some historical implementations do it.

System calls that update the time entry fields in the *stat* structure must be documented by the implementors. POSIX.1 conforming systems should not update the time entry fields for functions listed in POSIX.1 unless the standard requires that they do, except in the case of documented extensions to the standard.

Note that *st_dev* must be unique within a Local Area Network (LAN) in a “system” made up of multiple computers’ file systems connected by a LAN.

Networked implementations of a POSIX.1 system must guarantee that all files visible within the file tree (including parts of the tree that may be remotely mounted from other machines on the network) on each individual processor are uniquely identified by the combination of the *st_ino* and *st_dev* fields.

B.5.6.2 Get File Status

The intent of the paragraph describing “additional or alternate file access control mechanisms” is to allow a secure implementation where a process with a label that does not dominate the file’s label cannot perform a *stat()* function. This is not related to read permission; a process with a label that dominates the file’s

3213 label will not need read permission. An implementation that supports write-up
3214 operations could fail *fstat()* function calls even though it has a valid file descriptor
3215 open for writing.

3216 B.5.6.3 File Accessibility

3217 In early drafts of POSIX.1, some inadequacies in the *access()* function led to the
3218 creation of an *eaccess()* function because:

- 3219 (1) Historical implementations of *access()* do not test file access correctly
3220 when the process's real user ID is super-user. In particular, they always
3221 return zero when testing execute permissions without regard to whether
3222 the file is executable.
- 3223 (2) The super-user has complete access to all files on a system. As a conse-
3224 quence, programs started by the super-user and switched to the effective
3225 user ID with lesser privileges cannot use *access()* to test their file access
3226 permissions.

3227 However, the historical model of *eaccess()* does not resolve problem (1), so POSIX.1
3228 now allows *access()* to behave in the desired way because several implementa-
3229 tions have corrected the problem. It was also argued that problem (2) is more
3230 easily solved by using *open()*, *chdir()*, or one of the *exec* functions as appropriate
3231 and responding to the error, rather than creating a new function that would not
3232 be as reliable. Therefore, *eaccess()* was taken back out of POSIX.1.

3233 Secure implementations will probably need an extended *access()*-like function, but
3234 there were not enough of the requirements to define it yet. This could be pro-
3235 posed as an extension for a future amendment to POSIX.1.

3236 The sentence concerning appropriate privileges and execute permission bits
3237 reflects the two possibilities implemented by historical implementations when
3238 checking super-user access for *X_OK*.

3239 B.5.6.4 Change File Modes

3240 POSIX.1 specifies that the *S_ISGID* bit is cleared by *chmod()* on a regular file
3241 under certain conditions. This is specified on the assumption that regular files
3242 may be executed, and the system should prevent users from making executable
3243 *setgid* files perform with privileges that the caller does not have. On implementa-
3244 tions that support execution of other file types, the *S_ISGID* bit should be cleared
3245 for those file types under the same circumstances.

3246 Implementations that use the *S_ISUID* bit to indicate some other function (for
3247 example, mandatory record locking) on nonexecutable files need not clear this bit
3248 on writing. They should clear the bit for executable files and any other cases
3249 where the bit grants special powers to processes that change the file contents.
3250 Similar comments apply to the *S_ISGID* bit.

251 B.5.6.5 Change Owner and Group of File

252 System III and System V allow a user to give away files; that is, the owner of a file
253 may change its user ID to anything. This is a serious problem for implementa-
254 tions that are intended to meet government security regulations. Version 7 and
255 4.3BSD permit only the super-user to change the user ID of a file. Some govern-
256 ment agencies (usually not ones concerned directly with security) find this limita-
257 tion too confining. POSIX.1 uses “may” to permit secure implementations while
258 not disallowing System V.

259 System III and System V allow the owner of a file to change the group ID to any-
260 thing. Version 7 permits only the super-user to change the group ID of a file.
261 4.3BSD permits the owner to change the group ID of a file to its effective group ID
262 or to any of the groups in the list of supplementary group IDs, but to no others.

263 Although *chown()* can be used on some systems by the file owner to change the
264 owner and group to any desired values, the only portable use of this function is to
265 change the group of a file to the effective GID of the calling process or to a member
266 of its group set.

267 The decision to require that, for nonprivileged processes, the *S_ISUID* and
268 *S_ISGID* bits be cleared on regular files, but only *may* be cleared on nonregular
269 files, was to allow plans for using these bits in implementation-specified manners
270 on directories. Similar cases could be made for other file types, so POSIX.1 does
271 not require that these bits be cleared except on regular files. As these cases arise,
272 the system implementors will have to determine whether these features enable
273 any security loopholes and specify appropriate restrictions. If the implementation
274 supports executing any file types other than regular files, the *S_ISUID* and
275 *S_ISGID* bits should be cleared for those file types in the same way as they are on
276 regular files.

277 B.5.6.6 Set File Access and Modification Times

278 The *actime* structure member must be present so that an application may set it,
279 even though an implementation may ignore it and not change the access time on
280 the file. If an application intends to leave one of the times of a file unchanged
281 while changing the other, it should use *stat()* to retrieve the file’s *st_atime* and
282 *st_mtime* parameters, set *actime* and *modtime* in the buffer, and change one of
283 them before making the *utime()* call.

284 B.5.7 Configurable Pathname Variables

285 When the run-time facility described in B.4.8 was designed, it was realized that
286 some variables change depending on the file system. For example, it is quite
287 feasible for a system to have two varieties of file systems mounted: a System V
288 file system and a BSD “Fast File System.”

289 If limited to strictly compile-time features, no application that was widely distri-
290 buted in executable binary form could rely on more than 14 bytes in a pathname
291 component, as that is the minimum published for {NAME_MAX} in POSIX.1. The
292 *pathconf()* function allows the application to take advantage of the most liberal
293 file system available at run-time. In many BSD-based systems, 255 bytes are
294 allowed for pathname components.

3295 These values are potentially changeable at the directory level, not just at the file
3296 system. And, unlike the overall system variables, there is no guarantee that
3297 these might not change during program execution.

3298 **B.5.7.1 Get Configurable Pathname Variables**

3299 The *pathconf()* function was proposed immediately after the *sysconf()* function
3300 when it was realized that some configurable values may differ across file system,
3301 directory, or device boundaries.

3302 For example, {NAME_MAX} frequently changes between System V and BSD-based
3303 file systems; System V uses a maximum of 14, BSD 255. On an implementation
3304 that provided both types of file systems, an application would be forced to limit all
3305 pathname components to 14 bytes, as this would be the value specified in
3306 `<limits.h>` on such a system.

3307 Therefore, various useful values can be queried on any pathname or file descrip-
3308 tor, assuming that the appropriate permissions are in place.

3309 The value returned for the variable {PATH_MAX} indicates the longest relative
3310 pathname that could be given if the specified directory is the process's current
3311 working directory. A process may not always be able to generate a name that
3312 long and use it if a subdirectory in the pathname crosses into a more restrictive
3313 file system.

3314 The value returned for the variable {_POSIX_CHOWN_RESTRICTED} also applies
3315 to directories that do not have file systems mounted on them. The value may
3316 change when crossing a mount point, so applications that need to know should
3317 check for each directory. [An even easier check is to try the *chown()* function and
3318 look for an error in case it happens.]

3319 Unlike the values returned by *sysconf()*, the pathname-oriented variables are
3320 potentially more volatile and are not guaranteed to remain constant throughout
3321 the process's lifetime. For example, in between two calls to *pathconf()*, the file
3322 system in question may have been unmounted and remounted with different
3323 characteristics.

3324 Also note that most of the errors are optional. If one of the variables always has
3325 the same value on an implementation, the implementation need not look at *path*
3326 or *files* to return that value and is, therefore, not required to detect any of the
3327 errors except the meaning of [EINVAL] that indicates that the value of *name* is not
3328 valid for that variable.

3329 If the value of any of the limits described in 2.8.4 or 2.8.5 are indeterminate (logi-
3330 cally infinite), they will not be defined in `<limits.h>` and the *pathconf()* and
3331 *fpathconf()* functions will return -1 without changing *errno*. This can be dis-
3332 tinguished from the case of giving an unrecognized *name* argument because *errno*
3333 will be set to [EINVAL] in this case.

3334 Since -1 is a valid return value for the *pathconf()* and *fpathconf()* functions,
3335 applications should set *errno* to zero before calling them and check *errno* only if
3336 the return value is -1.

B.6 Input and Output Primitives

System III and System V have included a flag, `O_NDELAY`, to mark file descriptors so that user processes would not block when doing I/O to them. If the flag is set, a `read()` or `write()` call that would otherwise need to block for data returns a value of zero instead. But a `read()` call also returns a value of zero on end-of-file, and applications have no way to distinguish between these two conditions.

BSD systems support a similar feature through a flag with the same name, but somewhat different semantics. The flag applies to all users of a file (or socket) rather than only to those sharing a file descriptor. The BSD interface provides a solution to the problem of distinguishing between a blocking condition and an end-of-file condition by returning an error, `[EWOULDBLOCK]`, on a blocking condition.

The 1984 */usr/group Standard* (B59) includes an interface with some features from both System III/V and BSD. The overall semantics are that it applies only to a file descriptor. However, the return indication for a blocking condition is an error, `[EAGAIN]`. This was the starting point for POSIX.1.

The problem with the 1984 */usr/group Standard* (B59) is that it does not allow compatibility with existing applications. An implementation cannot both conform to that standard and support applications written for existing System V or BSD systems. Several changes have been considered address this issue. These include:

- (1) No change (from 1984 */usr/group Standard* (B59))
- (2) Changing to System III/V semantics
- (3) Changing to BSD semantics
- (4) Broadening POSIX.1 to allow conforming implementation a choice among these semantics
- (5) Changing the name of the flag from `O_NDELAY`
- (6) Changing to System III/V semantics and providing a new call to distinguish between blocking and end-of-file conditions

Alternative (5) was the consensus choice. The new name is `O_NONBLOCK`. This alternative allows a conforming implementation to provide backward compatibility at the source and/or object level with either System III/V or BSD systems (but POSIX.1 does not require or even suggest that this be done). It also allows a Conforming POSIX.1 Application Using Extensions the functionality to distinguish between blocking and end-of-file conditions, and to do so in as simple a manner as any of the alternatives. The greatest shortcoming was that it forces all existing System III/V and BSD applications that use this facility to be modified in order to strictly conform to POSIX.1. This same shortcoming applies to (1) and (4) as well, and it applies to one group of applications for (2), (3), and (6).

Systems may choose to implement both `O_NDELAY` and `O_NONBLOCK`, and there is no conflict as long as an application does not turn both flags on at the same time.

3379 See also the discussion of scope in B.6.5.1.

3380 **B.6.1 Pipes**

3381 An implementation that fails *write()* operations on *fildes*[0] or *read()*s on *fildes*[1]
3382 is not required. Historical implementations (Version 7 and System V) return the
3383 error [EBADF] in such cases. This allows implementations to set up a second pipe
3384 for full duplex operation at the same time. A conforming application that uses the
3385 *pipe()* function as described in POSIX.1 will succeed whether this second pipe is
3386 present or not.

3387 **B.6.1.1 Create an Inter-Process Channel**

3388 The wording carefully avoids using the verb “to open” in order to avoid any impli-
3389 cation of use of *open()*.

3390 See also B.6.4.2.

3391 **B.6.2 File Descriptor Manipulation**

3392 **B.6.2.1 Duplicate an Open File Descriptor**

3393 The *dup()* and *dup2()* functions are redundant. Their services are also provided
3394 by the *fcntl()* function. They have been included in POSIX.1 primarily for histori-
3395 cal reasons, since many existing applications use them.

3396 While the brief code segment shown is very similar in behavior to *dup2()*, a con-
3397 forming implementation based on other functions defined by POSIX.1 is
3398 significantly more complex. Least obvious is the possible effect of a signal-
3399 catching function that could be invoked between steps and allocate or deallocate
3400 file descriptors. This could be avoided by blocking signals.

3401 The *dup2()* function is not marked obsolescent because it presents a type-safe ver-
3402 sion of functionality provided in a type-unsafe version by *fcntl()*. It is used in the
3403 current draft of the Ada binding to POSIX.1.

3404 The *dup2()* function is not intended for use in critical regions as a synchroniza-
3405 tion mechanism.

3406 In the description of [EBADF], the case of *fildes* being out of range is covered by
3407 the given case of *fildes* not being valid. The descriptions for *fildes* and *fildes2* are
3408 different because the only kind of invalidity that is relevant for *fildes2* is whether
3409 it is out of range; that is, it does not matter whether *fildes2* refers to an open file
3410 when the *dup2()* call is made.

3411 If *fildes2* is a valid file descriptor, it shall be closed, regardless of whether the
3412 function returns an indication of success or failure, unless *fildes2* is equal to
3413 *fildes*.

414 B.6.3 File Descriptor Deassignment

415 B.6.3.1 Close a File

416 Once a file is closed, the file descriptor no longer exists, since the integer
417 corresponding to it no longer refers to a file.

418 The use of interruptible device close routines should be discouraged to avoid prob-
419 lems with the implicit closes of file descriptors by *exec* and *exit*(). POSIX.1 only
420 intends to permit such behavior by specifying the [EINTR] error case.

421 B.6.4 Input and Output

422 The use of I/O with large byte counts has always presented problems. Ideas such
423 as *lread*() and *lwrite*() (using and returning *long*s) were considered at one time.
424 The current solution is to use abstract types on the C Standard [2] interface to
425 *read*() and *write*() (and not to discuss common usage). The abstract types can be
426 declared so that existing interfaces work, but can also be declared so that larger
427 types can be represented in future implementations. It is presumed that what-
428 ever constraints limit the maximum range of *size_t* also limit portable I/O requests
429 to the same range. POSIX.1 also limits the range further by requiring that the
430 byte count be limited so that a signed return value remains meaningful. Since
431 the return type is also a (signed) abstract type, the byte count can be defined by
432 the implementation to be larger than an *int* can hold.

433 POSIX.1 requires that no action be taken when *nbyte* is zero. This is not intended
434 to take precedence over detection of errors (such as invalid buffer pointers or file
435 descriptors). This is consistent with the rest of POSIX.1, but the phrasing here
436 could be misread to require detection of the zero case before any other errors. A
437 value of zero is to be considered a correct value, for which the semantics are a
438 no-op.

439 There were recommendations to add format parameters to *read*() and *write*() in
440 order to handle networked transfers among heterogeneous file system and base
441 hardware types. Such a facility may be required for support by the OSI presenta-
442 tion of layer services. However, it was determined that this should correspond
443 with similar C Language facilities, and that is beyond the scope of POSIX.1. The
444 concept was suggested to the developers of the C Standard [2] for their considera-
445 tion as a possible area for future work.

446 In 4.3BSD, a *read*() or *write*() that is interrupted by a signal before transferring
447 any data does not by default return an [EINTR] error, but is restarted. In 4.2BSD,
448 4.3BSD, and the Eighth Edition there is an additional function, *select*(), whose
449 purpose is to pause until specified activity (data to read, space to write, etc.) is
450 detected on specified file descriptors. It is common in applications written for
451 those systems for *select*() to be used before *read*() in situations (such as keyboard
452 input) where interruption of I/O due to a signal is desired. But this approach does
453 not conform, because *select*() is not in POSIX.1. 4.3BSD semantics can be provided
454 by extensions to POSIX.1.

455 POSIX.1 permits *read*() and *write*() to return the number of bytes successfully
456 transferred when interrupted by an error. This is not simply required because it

3457 was not done by Version 7, System III, or System V, and because some hardware
 3458 may not be capable of returning information about partial transfers if a device
 3459 operation is interrupted. Unfortunately, this does make writing a Conforming
 3460 POSIX.1 Application more difficult in circumstances where this could occur.

3461 Requiring this behavior does not address the situation of pipelined buffers, such
 3462 as might be found in streaming tape drives or other devices that read ahead of the
 3463 actual requests. The signal interruption will often indicate an exceptional condi-
 3464 tion and flush all buffers. Thus, the amount read from the device may be dif-
 3465 ferent from the amount transferred to the application.

3466 The issue of which files or file types are interruptible is considered an implemen-
 3467 tation design issue. This is often affected primarily by hardware and reliability
 3468 issues.

3469 There are no references to actions taken following an “unrecoverable error.” It is
 3470 considered beyond the scope of POSIX.1 to describe what happens in the case of
 3471 hardware errors.

3472 **B.6.4.1 Read from a File**

3473 POSIX.1 does not specify the value of the file offset after an error is returned;
 3474 there are too many cases. For programming errors, such as [EBADF], the concept
 3475 is meaningless since no file is involved. For errors that are detected immediately,
 3476 such as [EAGAIN], clearly the pointer should not change. After an interrupt or
 3477 hardware error, however, an updated value would be very useful and is the
 3478 behavior of many implementations.

3479 Note that a *read()* of zero bytes does not modify *st_atime*. A *read()* that requests
 3480 more than zero bytes, but returns zero, does modify *st_atime*. |

3481 **B.6.4.2 Write to a File**

3482 An attempt to write to a pipe or FIFO has several major characteristics:

3483 **Atomic/nonatomic**

3484 A write is atomic if the whole amount written in one operation is not
 3485 interleaved with data from any other process. This is useful when there
 3486 are multiple writers sending data to a single reader. Applications need
 3487 to know how large a write request can be expected to be performed atomi-
 3488 cally. This maximum is called {PIPE_BUF}. POSIX.1 does not say
 3489 whether write requests for more than {PIPE_BUF} bytes will be atomic,
 3490 but requires that writes of {PIPE_BUF} or fewer bytes shall be atomic.

3491 **Blocking/immediate**

3492 Blocking is only possible with O_NONBLOCK clear. If there is enough
 3493 space for all the data requested to be written immediately, the implemen-
 3494 tation should do so. Otherwise, the process may block; that is, pause
 3495 until enough space is available for writing. The effective size of a pipe or
 3496 FIFO (the maximum amount that can be written in one operation without
 3497 blocking) may vary dynamically, depending on the implementation, so it
 3498 is not possible to specify a fixed value for it.

```

99      Complete/partial/deferred
00      A write request,
01          int fildes;
02          size_t nbyte;
03          ssize_t ret;
04          char *buf;
05
06          ret = write (fildes, buf, nbyte);
07
08      may return
09
10      complete: ret = nbyte
11
12      partial:  ret < nbyte
13              This shall never happen if nbyte ≤ {PIPE_BUF}. If it does
14              happen (with nbyte > {PIPE_BUF}), POSIX.1 does not
15              guarantee atomicity, even if ret ≤ {PIPE_BUF}, because
16              atomicity is guaranteed according to the amount requested,
17              not the amount written.
18
19      deferred: ret = -1, errno = [EAGAIN]
20              This error indicates that a later request may succeed. It
21              does not indicate that it shall succeed, even if nbyte ≤
22              {PIPE_BUF}, because if no process reads from the pipe or
23              FIFO, the write will never succeed. An application could
24              usefully count the number of times [EAGAIN] is caused by a
25              particular value of nbyte > {PIPE_BUF} and perhaps do
26              later writes with a smaller value, on the assumption that
27              the effective size of the pipe may have decreased.

```

Partial and deferred writes are only possible with O_NONBLOCK set.

The relations of these properties are shown in the following tables.

Write to a Pipe or FIFO with O_NONBLOCK <i>clear</i>			
Immediately Writable:	None	Some	<i>nbyte</i>
<i>nbyte</i> ≤ {PIPE_BUF}	Atomic blocking <i>nbyte</i>	Atomic blocking <i>nbyte</i>	Atomic immediate <i>nbyte</i>
<i>nbyte</i> > {PIPE_BUF}	Blocking <i>nbyte</i>	Blocking <i>nbyte</i>	Blocking <i>nbyte</i>

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.

3537
3538
3539
3540
3541
3542
3543
3544

Write to a Pipe or FIFO with O_NONBLOCK set			
Immediately Writable:	None	Some	<i>nbyte</i>
$nbyte \leq \{PIPE_BUF\}$	-1, [EAGAIN]	-1, [EAGAIN]	Atomic <i>nbyte</i>
$nbyte > \{PIPE_BUF\}$	-1, [EAGAIN]	$< nbyte$ or -1, [EAGAIN]	$\leq nbyte$ or -1, [EAGAIN]

3545 There is no exception regarding partial writes when O_NONBLOCK is set. With
3546 the exception of writing to an empty pipe, POSIX.1 does not specify exactly when a
3547 partial write will be performed since that would require specifying internal
3548 details of the implementation. Every application should be prepared to handle
3549 partial writes when O_NONBLOCK is set and the requested amount is greater
3550 than {PIPE_BUF}, just as every application should be prepared to handle partial
3551 writes on other kinds of file descriptors.

3552 The intent of forcing writing at least one byte if any can be written is to assure
3553 that each write will make progress if there is any room in the pipe. If the pipe is
3554 empty, {PIPE_BUF} bytes must be written; if not, at least some progress must
3555 have been made.

3556 Where POSIX.1 requires -1 to be returned and *errno* set to [EAGAIN], most historical
3557 implementations return zero (with the O_NDELAY flag set—that flag is the
3558 historical predecessor of O_NONBLOCK, but is not itself in POSIX.1). The error
3559 indications in POSIX.1 were chosen so that an application can distinguish these
3560 cases from end-of-file. While *write()* cannot receive an indication of end-of-file,
3561 *read()* can, and the two functions have similar return values. Also, some existing
3562 systems (e.g., Eighth Edition) permit a write of zero bytes to mean that the reader
3563 should get an end-of-file indication; for those systems, a return value of zero from
3564 *write()* indicates a successful write of an end-of-file indication.

3565 The concept of a {PIPE_MAX} limit (indicating the maximum number of bytes that
3566 can be written to a pipe in a single operation) was considered, but rejected,
3567 because this concept would unnecessarily limit application writing.

3568 See also the discussion of O_NONBLOCK in B.6.

3569 Writes can be serialized with respect to other reads and writes. If a *read()* of file
3570 data can be proven (by any means) to occur after a *write()* of the data, it must
3571 reflect that *write()*, even if the calls are made by different processes. A similar
3572 requirement applies to multiple write operations to the same file position. This is
3573 needed to guarantee the propagation of data from *write()* calls to subsequent
3574 *read()* calls. This requirement is particularly significant for networked file sys-
3575 tems, where some caching schemes violate these semantics.

3576 Note that this is specified in terms of *read()* and *write()*. Additional calls such as
3577 the common *readv()* and *writv()* would want to obey these semantics. A new
3578 “high-performance” write analog that did not follow these serialization require-
3579 ments would also be permitted by this wording. POSIX.1 is also silent about any
3580 effects of application-level caching (such as that done by *stdio*).

POSIX.1 does not specify the value of the file offset after an error is returned; there are too many cases. For programming errors, such as [EBADF], the concept is meaningless since no file is involved. For errors that are detected immediately, such as [EAGAIN], clearly the pointer should not change. After an interrupt or hardware error, however, an updated value would be very useful and is the behavior of many implementations.

POSIX.1 does not specify behavior of concurrent writes to a file from multiple processes. Applications should use some form of concurrency control.

B.6.5 Control Operations on Files

B.6.5.1 Data Definitions for File Control Operations

The main distinction between the file descriptor flags and the file status flags is scope. The former apply to a single file descriptor only, while the latter apply to all file descriptors that share a common open file description [by inheritance through *fork()* or an *F_DUPFD* operation with *fcntl()*]. For *O_NONBLOCK*, this scoping is like that of *O_NDELAY* in System V rather than in 4.3BSD, where the scoping for *O_NDELAY* is different from all the other flags accessed via the same commands.

For example:

```
fd1 = open (pathname, oflags);
fd2 = dup (fd1);
fd3 = open (pathname, oflags);
```

Does an *fcntl()* call on *fd1* also apply to *fd2* or *fd3* or to both? According to POSIX.1, *F_SETFD* applies only to *fd1*, while *F_SETFL* applies to *fd1* and *fd2* but not to *fd3*. This is in agreement with all common historical implementations except for BSD with the *F_SETFL* command and the *O_NDELAY* flag (which would apply to *fd3* as well). Note that this does not force any incompatibilities in BSD implementations, because *O_NDELAY* is not in POSIX.1. See also B.6.

Historically, the file descriptor flags have had only the literal values 0 and 1. POSIX.1 defines the symbolic name *FD_CLOEXEC* to permit a more graceful extension of this functionality. Owners of existing applications should be aware of the need to change applications using the literal values, and implementors should be aware of the existence of this practice in existing applications.

B.6.5.2 File Control

The ellipsis in the Synopsis is the syntax specified by the C Standard [2] for a variable number of arguments. It is used because System V uses pointers for the implementation of file locking functions.

The *arg* values to *F_GETFD*, *F_SETFD*, *F_GETFL*, and *F_SETFL* all represent flag values to allow for future growth. Applications using these functions should do a read-modify-write operation on them, rather than assuming that only the values defined by POSIX.1 are valid. It is a common error to forget this, particularly in the case of *F_SETFD*, because there is only one flag in POSIX.1.

3622 POSIX.1 permits concurrent read and write access to file data using the *fcntl()*
 3623 function; this is a change from the 1984 */usr/group Standard* (B59) and early
 3624 POSIX.1 drafts, which included a *lockf()* function. Without concurrency controls,
 3625 this feature may not be fully utilized without occasional loss of data. Since other
 3626 mechanisms for creating critical regions, such as semaphores, are not included, a
 3627 file record locking mechanism was thought to be appropriate. The *fcntl()* mechan-
 3628 ism may be used to implement semaphores, although access is not first-in-first-
 3629 out without extra application development effort.

3630 Data losses occur in several ways. One is that read and write operations are not
 3631 atomic, and as such a reader may get segments of new and old data if con-
 3632 currently written by another process. Another occurs when several processes try
 3633 to update the same record, without sequencing controls; several updates may
 3634 occur in parallel and the last writer will “win.” Another case is a b-tree or other
 3635 internal list-based database that is undergoing reorganization. Without exclusive
 3636 use to the tree segment by the updating process, other reading processes chance
 3637 getting lost in the database when the index blocks are split, condensed, inserted,
 3638 or deleted. While *fcntl()* is useful for many applications, it is not intended to be
 3639 overly general and will not handle the b-tree example well.

3640 This facility is only required for regular files because it is not appropriate for
 3641 many devices such as terminals and network connections.

3642 Since *fcntl()* works with “any file descriptor associated with that file, however it is
 3643 obtained,” the file descriptor may have been inherited through a *fork()* or *exec*
 3644 operation and thus may affect a file that another process also has open.

3645 The use of the open file description to identify what to lock requires extra calls
 3646 and presents problems if several processes are sharing an open file description,
 3647 but there are too many implementations of the existing mechanism for POSIX.1 to
 3648 use different specifications.

3649 Another consequence of this model is that closing any file descriptor for a given
 3650 file (whether or not it is the same open file description that created the lock)
 3651 causes the locks on that file to be relinquished for that process. Equivalently, any
 3652 close for any file/process pair relinquishes the locks owned on that file for that
 3653 process. But note that while an open file description may be shared through
 3654 *fork()*, locks are not inherited through *fork()*. Yet locks may be inherited through
 3655 one of the *exec* functions.

3656 The identification of a machine in a network environment is outside of the scope
 3657 of POSIX.1. Thus, an *l_sysid* member, such as found in System V, is not included
 3658 in the locking structure.

3659 Since locking is performed with *fcntl()*, rather than *lockf()*, this specification
 3660 prohibits use of advisory exclusive locking on a file that is not open for writing.

3661 Before successful return from a *F_SETLK* or *F_SETLKW* request, the previous lock
 3662 type for each byte in the specified region shall be replaced by the new lock type.
 3663 This can result in a previously locked region being split into smaller regions. If
 3664 this would cause the number of regions being held by all processes in the system
 3665 to exceed a system-imposed limit, the *fcntl()* function returns *-1* with *errno* set to
 3666 *[ENOLCK]*.

Mandatory locking was a major feature of the 1984 */usr/group Standard* (B59). For advisory file record locking to be effective, all processes that have access to a file must cooperate and use the advisory mechanism before doing I/O on the file. Enforcement-mode record locking is important when it cannot be assumed that all processes are cooperating. For example, if one user uses an editor to update a file at the same time that a second user executes another process that updates the same file and if only one of the two processes is using advisory locking, the processes are not cooperating. Enforcement-mode record locking would protect against accidental collisions.

Secondly, advisory record locking requires a process using locking to bracket each I/O operation with lock (or test) and unlock operations. With enforcement-mode file and record locking, a process can lock the file once and unlock when all I/O operations have been completed. Enforcement-mode record locking provides a base that can be enhanced, for example, with sharable locks. That is, the mechanism could be enhanced to allow a process to lock a file so other processes could read it, but none of them could write it.

Mandatory locks were omitted for several reasons:

- (1) Mandatory lock setting was done by multiplexing the set-group-ID bit in most implementations; this was confusing, at best.
- (2) The relationship to file truncation as supported in 4.2BSD was not well specified.
- (3) Any publicly readable file could be locked by anyone. Many historical implementations keep the password database in a publicly readable file. A malicious user could thus prohibit logins. Another possibility would be to hold open a long-distance telephone line.
- (4) Some demand-paged historical implementations offer memory mapped files, and enforcement cannot be done on that type of file.

Since sleeping on a region is interrupted with any signal, *alarm()* may be used to provide a timeout facility in applications requiring it. This is useful in deadlock detection. Because implementation of full deadlock detection is not always feasible, the [EDEADLK] error was made optional.

B.6.5.3 Reposition Read/Write File Offset

The C Standard (2) includes the functions *fgetpos()* and *fsetpos()*, which work on very large files by use of a special positioning type.

Although *lseek()* may position the file offset beyond the end of the file, this function does not itself extend the size of the file. While the only function in POSIX.1 that may extend the size of the file is *write()*, several C Standard (2) functions, such as *fwrite()*, *fprintf()*, etc., may do so [by causing calls on *write()*].

An invalid file offset that would cause [EINVAL] to be returned may be both implementation defined and device dependent (for example, memory may have few invalid values). A negative file offset may be valid for some devices in some implementations.

3710 See B.6.5.2 for an explanation of the use of signed and unsigned offsets with
3711 *lseek()*.

3712 B.7 Device- and Class-Specific Functions

3713 There were several sources of difficulties involved with using historical interfaces
3714 as the basis of this section:

- 3715 (1) The basic Version 7 *ioctl()* mechanism is difficult to specify adequately,
3716 due to its use of a third argument that varies in both size and type
3717 according to the second, command, argument.
- 3718 (2) System III introduced and System V continued *ioctl()* commands that are
3719 completely different from those of Version 7.
- 3720 (3) 4.2BSD and other BSD systems added to the basic Version 7 *ioctl()* com-
3721 mand set; some of these were for features such as job control that
3722 POSIX.1 eventually adopted.
- 3723 (4) None of the basic historical implementations are adequate in an interna-
3724 tional environment. This concern is not technically within the scope of
3725 POSIX.1, but the goal of POSIX.1 was to mandate no unnecessary impedi-
3726 ments to internationalization.

3727 The 1984 */usr/group Standard* [B59] attempted to specify a portable mechanism
3728 that application writers could use to get and set the modes of an asynchronous
3729 terminal. The intention of that committee was to provide an interface that was
3730 neither implementation specific nor hardware dependent. Initial proposals dealt
3731 with high-level routines similar to the *curses* library (available on most historical
3732 implementations). In such an implementation, the user interface would consist of
3733 calls similar to:

```
3734     setraw();  
3735     setcooked();
```

3736 It was quickly pointed out that if such routines were standardized, the definition
3737 of “raw” and “cooked” would have to be provided. If these modes were not well
3738 defined in POSIX.1, application code could not be written in a portable way. How-
3739 ever, the definition of the terms would force low-level concepts to be included in a
3740 supposedly high-level interface definition.

3741 Focus was given to the necessary low-level attributes that were needed to support
3742 the necessary terminal characteristics (e.g., line speeds, raw mode, cooked mode,
3743 etc.). After considerable debate, a structure similar to, but more flexible than, the
3744 System III *termio* was accepted. The format of that structure, referred to as the
3745 *termios* structure, has formed the basis for the current section.

3746 A method was needed to communicate with the system about the *termios* informa-
3747 tion. Proposals included:

- 3748 (1) The *ioctl()* function as in System V. This had the same problems as men-
3749 tioned previously for the Version 7 *ioctl()* function and was basically
3750 identical to it. Another problem was that the direction of the command
3751 (whether information is written from or read into the third argument)

was not specified—in historical implementations, only the device driver knows this information. This was a problem for networked implementations. It was also a problem that there was no size parameter to specify the variable size of the third argument, and there was a similar problem with its type.

- (2) An *iocntl()* function with additional arguments specifying direction, type, and size. But these new arguments did not help application writers, who would have no control over their values, which would have to match each command exactly. The new arguments did, however, solve the problems of networked implementations. And *iocntl()* would have been implementable in terms of *ioctl()* on historical implementations (without need for modifying existing code), although it would have been easy to update existing code to use the arguments directly.
- (3) A *termcntl()* function with the same arguments as proposed for the *iocntl()* function. The difference was that *termcntl()* would be limited to terminal interface functions; there would be other interface functions, such as a *tapecntl()* function for tape interfaces, rather than a single general device interface routine.
- (4) Unspecified functions. The issue of what the interface function(s) should be called was avoided for many of the early drafts while details of the information to be handled was of prime concern. The resulting specification resembled the information in System V, but attempted to avoid problems of case, speed, networks, and internationalization.

Specific *tc*()* functions³⁾ to replace each *ioctl()* function were finally incorporated into POSIX.1, instead of any of the previously mentioned proposals.

The issue of modem control was excluded from POSIX.1 on the grounds that

- It was concerned with setting and control of hardware timers.
- The appropriate timers and settings vary widely internationally.
- Feedback from European computer manufacturers indicated that this facility was not consistent with European needs and that specification of such a facility was not a requirement for portability.

B.7.1 General Terminal Interface

If the implementation does not support this interface on any device types, it should behave as if it were being used on a device that is not a terminal device (in most cases *errno* will be set to [ENOTTY]) on return from functions defined by this interface. This is based on the fact that many applications are written to run both interactively and in some noninteractive mode, and they adapt themselves at run time. Requiring that they all be modified to test an environment variable to

3) The notation *tc*()* is reminiscent of shell pattern matching notation and is an abbreviated way of referring to all functions beginning with the letters “tc.”

3792 determine if they should try to adapt is unnecessary. On a system that provides
 3793 no Section 7 interface, providing all the entry points as stubs that return
 3794 [ENOTTY] (or an equivalent, as appropriate) has the same effect and requires no
 3795 changes to the application.

3796 Although the needs of both interface implementors and application developers
 3797 were addressed throughout POSIX.1, this section pays more attention to the needs
 3798 of the latter. This is because, while many aspects of the programming interface
 3799 can be hidden from the user by the application developer, the terminal interface is
 3800 usually a large part of the user interface. Although to some extent the application
 3801 developer can build missing features or work around inappropriate ones, the
 3802 difficulties of doing that are greater in the terminal interface than elsewhere. For
 3803 example, efficiency prohibits the average program from interpreting every character
 3804 passing through it in order to simulate character erase, line kill, etc. These
 3805 functions should usually be done by the operating system, possibly at the interrupt
 3806 level.

3807 The *tc**() functions were introduced as a way of avoiding the problems inherent in
 3808 the traditional *ioclt*() function and in variants of it that were proposed. For example,
 3809 *tcsetattr*() is specified in place of the use of the TCSETA *ioclt*() command function.
 3810 This allows specification of all the arguments in a manner consistent with
 3811 the C Standard [2], unlike the varying third argument of *ioclt*(), which is sometimes
 3812 a pointer (to any of many different types) and sometimes an *int*.

3813 The advantages of this new method include:

- 3814 — It allows strict type checking.
- 3815 — The direction of transfer of control data is explicit.
- 3816 — Portable capabilities are clearly identified.
- 3817 — The need for a general interface routine is avoided.
- 3818 — Size of the argument is well-defined (there is only one type).

3819 The disadvantages include:

- 3820 — No historical implementation uses the new method.
- 3821 — There are many small routines instead of one general-purpose one.
- 3822 — The historical parallel with *fcntl*() is broken.

3823 B.7.1.1 Interface Characteristics

3824 B.7.1.1.1 Opening a Terminal Device File

3825 Further implications of the effects of CLOCAL are discussed in 7.1.2.4.

3826 B.7.1.1.2 Process Groups

3827 There is a potential race when the members of the foreground process group on a
 3828 terminal leave that process group, either by exit or by changing process groups.
 3829 After the last process exits the process group, but before the foreground process
 3830 group ID of the terminal is changed (usually by a job-control shell), it would be
 3831 possible for a new process to be created with its process ID equal to the terminal's

foreground process group ID. That process might then become the process group leader and accidentally be placed into the foreground on a terminal that was not necessarily its controlling terminal. As a result of this problem, the controlling terminal is defined to not have a foreground process group during this time.

The cases where a controlling terminal has no foreground process group occur when all processes in the foreground process group either terminate and are waited for or join other process groups via *setpgid()* or *setsid()*. If the process group leader terminates, this is the first case described; if it leaves the process group via *setpgid()*, this is the second case described [a process group leader cannot successfully call *setsid()*]. When one of those cases causes a controlling terminal to have no foreground process group, it has two visible effects on applications. The first is the value returned by *tcgetpgrp()*, as discussed in 7.2.3 and B.7.2.3. The second (which occurs only in the case where the process group leader terminates) is the sending of signals in response to special input characters. The intent of POSIX.1 is that no process group be wrongly identified as the foreground process group by *tcgetpgrp()* or unintentionally receive signals because of placement into the foreground.

In 4.3BSD, the old process group ID continues to be used to identify the foreground process group and is returned by the function equivalent to *tcgetpgrp()*. In that implementation it is possible for a newly created process to be assigned the same value as a process ID and then form a new process group with the same value as a process group ID. The result is that the new process group would receive signals from this terminal for no apparent reason, and POSIX.1 precludes this by forbidding a process group from entering the foreground in this way. It would be more direct to place part of the requirement made by the last sentence under 3.1.1, but there is no convenient way for that subclause to refer to the value that *tcgetpgrp()* returns, since in this case there is no process group and thus no process group ID.

One possibility for a conforming implementation is to behave similarly to 4.3BSD, but to prevent this reuse of the ID, probably in the implementation of *fork()*, as long as it is in use by the terminal.

Another possibility is to recognize when the last process stops using the terminal's foreground process group ID, which is when the process group lifetime ends, and to change the terminal's foreground process group ID to a reserved value that is never used as a process ID or process group ID. (See the definition of *process group lifetime* in 2.2.2.) The process ID can then be reserved until the terminal has another foreground process group.

The 4.3BSD implementation permits the leader (and only member) of the foreground process group to leave the process group by calling the equivalent of *setpgid()* and to later return, expecting to return to the foreground. There are no known application needs for this behavior, and POSIX.1 neither requires nor forbids it (except that it is forbidden for session leaders) by leaving it unspecified.

B.7.1.1.3 The Controlling Terminal

POSIX.1 does not specify a mechanism by which to allocate a controlling terminal. This is normally done by a system utility (such as *getty*) and is considered an administrative feature outside the scope of POSIX.1.

3878 Historical implementations allocate controlling terminals on certain *open()* calls.
3879 Since *open()* is part of POSIX.1, its behavior had to be dealt with. The traditional
3880 behavior is not required because it is not very straightforward or flexible for
3881 either implementations or applications. However, because of its prevalence, it
3882 was not practical to disallow this behavior either. Thus, a mechanism was stand-
3883 ardized to ensure portable, predictable behavior in *open()*.

3884 Some historical implementations deallocate a controlling terminal on its last sys-
3885 temwide close. This behavior is neither required nor prohibited. Even on imple-
3886 mentations that do provide this behavior, applications generally cannot depend on
3887 it due to its systemwide nature.

3888 B.7.1.1.4 Terminal Access Control

3889 The access controls described in this subclause apply only to a process that is
3890 accessing its controlling terminal. A process accessing a terminal that is not its
3891 controlling terminal is effectively treated the same as a member of the foreground
3892 process group. While this may seem unintuitive, note that these controls are for
3893 the purpose of job control, not security, and job control relates only to a process's
3894 controlling terminal. Normal file access permissions handle security.

3895 If the process calling *read()* or *write()* is in a background process group that is
3896 orphaned, it is not desirable to stop the process group, as it is no longer under the
3897 control of a job-control shell that could put it into foreground again. Accordingly,
3898 calls to *read()* or *write()* functions by such processes receive an immediate error
3899 return. This is different than in 4.2BSD, which kills orphaned processes that
3900 receive terminal stop signals.

3901 The foreground/background/orphaned process group check performed by the ter-
3902 minal driver must be repeatedly performed until the calling process moves into
3903 the foreground or until the process group of the calling process becomes orphaned.
3904 That is, when the terminal driver determines that the calling process is in the
3905 background and should receive a job-control signal, it sends the appropriate sig-
3906 nal (SIGTTIN or SIGTTOU) to every process in the process group of the calling pro-
3907 cess and then it allows the calling process to immediately receive the signal. The
3908 latter is typically performed by blocking the process so that the signal is immedi-
3909 ately noticed. Note, however, that after the process finishes receiving the signal
3910 and control is returned to the driver, the terminal driver must reexecute the fore-
3911 ground/background/orphaned process group check. The process may still be in
3912 the background, either because it was continued in the background by a job-
3913 control shell, or because it caught the signal and did nothing.

3914 The terminal driver repeatedly performs the foreground/background/orphaned
3915 process group checks whenever a process is about to access the terminal. In the
3916 case of *write()* or the control functions in 7.2, the check is performed at the entry
3917 of the function. In the case of *read()*, the check is performed not only at the entry
3918 of the function, but also after blocking the process to wait for input characters (if
3919 necessary). That is, once the driver has determined that the process calling the
3920 *read()* function is in the foreground, it attempts to retrieve characters from the
3921 input queue. If the queue is empty, it blocks the process waiting for characters.
3922 When characters are available and control is returned to the driver, the terminal
3923 driver must return to the repeated foreground/background/orphaned process
3924 group check again. The process may have moved from the foreground to the

background while it was blocked waiting for input characters.

B.7.1.1.5 Input Processing and Reading Data

There is no additional rationale provided for this subclause.

B.7.1.1.6 Canonical Mode Input Processing

The term “character” is intended here. ERASE should erase the last character, not the last byte. In the case of multibyte characters, these two may be different.

4.3BSD has a WERASE character that erases the last “word” typed (but not any preceding blanks or tabs). A word is defined as a sequence of nonblank characters, with tabs counted as blanks. Like ERASE, WERASE does not erase beyond the beginning of the line. This WERASE feature has not been specified in POSIX.1 because it is difficult to define in the international environment. It is only useful for languages where words are delimited by blanks. In some ideographic languages, such as Japanese and Chinese, words are not delimited at all. The WERASE character should presumably take one back to the beginning of a sentence in those cases; practically, this means it would not get much use for those languages.

It should be noted that there is a possible inherent deadlock if the application and implementation conflict on the value of MAX_CANON. With ICANON set (if IXOFF is enabled) and more than MAX_CANON characters transmitted without a linefeed, transmission will be stopped, the linefeed (or carriage return when ICRLF is set) will never arrive, and the `read()` will never be satisfied.

An application should not set IXOFF if it is using canonical mode unless it knows that (even in the face of a transmission error) the conditions described previously cannot be met or unless it is prepared to deal with the possible deadlock in some other way, such as timeouts.

It should also be noted that this can be made to happen in noncanonical mode if the trigger value for sending IXOFF is less than VMIN and VTIME is zero.

B.7.1.1.7 Noncanonical Mode Input Processing

Some points to note about MIN and TIME:

- (1) The interactions of MIN and TIME are not symmetric. For example, when MIN > 0 and TIME = 0, TIME has no effect. However, in the opposite case where MIN = 0 and TIME > 0, both MIN and TIME play a role in that MIN is satisfied with the receipt of a single character.
- (2) Also note that in case A (MIN > 0, TIME > 0), TIME represents an inter-character timer while in case C (MIN = 0, TIME > 0) TIME represents a read timer.

These two points highlight the dual purpose of the MIN/TIME feature. Cases A and B, where MIN > 0, exist to handle burst-mode activity (e.g., file transfer programs) where a program would like to process at least MIN characters at a time. In case A, the intercharacter timer is activated by a user as a safety measure; in case B, it is turned off.

3966 Cases C and D exist to handle single-character timed transfers. These cases are
 3967 readily adaptable to screen-based applications that need to know if a character is
 3968 present in the input queue before refreshing the screen. In case C the read is
 3969 timed; in case D, it is not.

3970 Another important note is that MIN is always just a minimum. It does not denote
 3971 a record length. That is, if a program does a read of 20 bytes, MIN is 10, and 25
 3972 characters are present, 20 characters shall be returned to the user. In the special
 3973 case of MIN=0, this still applies: if more than one character is available, they all
 3974 will be returned immediately.

3975 **B.7.1.1.8 Writing Data and Output Processing**

3976 There is no additional rationale provided for this subclause.

3977 **B.7.1.1.9 Special Characters**

3978 There is no additional rationale provided for this subclause.

3979 **B.7.1.1.10 Modem Disconnect**

3980 There is no additional rationale provided for this subclause.

3981 **B.7.1.1.11 Closing a Terminal Device File**

3982 POSIX.1 is silent on whether a *close()* will block on waiting for transmission to
 3983 drain, or even if a *close()* might cause a flush of pending output. If the application
 3984 is concerned about this, it should call the appropriate function, such as *tcdrain()*,
 3985 to ensure the desired behavior.

3986 **B.7.1.2 Parameters That Can Be Set**

3987 **B.7.1.2.1 *termios* Structure**

3988 This structure is part of an interface that, in general, retains the historic group-
 3989 ing of flags. Although a more optimal structure for implementations may be pos-
 3990 sible, the degree of change to applications would be significantly larger.

3991 **B.7.1.2.2 Input Modes**

3992 Some historical implementations treated a long break as multiple events, as
 3993 many as one per character time. The wording in POSIX.1 explicitly prohibits this.

3994 Although the ISTRIP flag is normally superfluous with today's terminal hardware
 3995 and software, it is historically supported. Therefore, applications may be using
 3996 ISTRIP, and there is no technical problem with supporting this flag. Also, applica-
 3997 tions may wish to receive only 7-bit input bytes and may not be connected directly
 3998 to the hardware terminal device (for example, when a connection traverses a
 3999 network).

4000 Also, there is no requirement in general that the terminal device ensures that
 4001 high-order bits beyond the specified character size are cleared. ISTRIP provides
 4002 this function for 7-bit characters, which are common.

In dealing with multibyte characters, the consequences of a parity error in such a character, or in an escape sequence affecting the current character set, are beyond the scope of POSIX.1 and are best dealt with by the application processing the multibyte characters.

B.7.1.2.3 Output Modes

POSIX.1 does not describe postprocessing of output to a terminal or detailed control of that from a portable application. (That is, translation of newline to carriage return followed by linefeed or tab processing.) There is nothing that a portable application should do to its output for a terminal because that would require knowledge of the operation of the terminal. It is the responsibility of the operating system to provide postprocessing appropriate to the output device, whether it is a terminal or some other type of device.

Extensions to POSIX.1 to control the type of postprocessing already exist and are expected to continue into the future. The control of these features is primarily to adjust the interface between the system and the terminal device so the output appears on the display correctly. This should be set up before use by any application.

In general, both the input and output modes should not be set absolutely, but rather modified from the inherited state.

B.7.1.2.4 Control Modes

This subclause could be misread that the symbol “CSIZE” is a title in Table 7-3. Although it does serve that function, it is also a required symbol, as a literal reading of POSIX.1 (and the caveats about typography) would indicate.

B.7.1.2.5 Local Modes

Noncanonical mode is provided to allow fast bursts of input to be read efficiently while still allowing single-character input.

The ECHONL function historically has been in many implementations. Since there seems to be no technical problem with supporting ECHONL, it is included in POSIX.1 to increase consensus.

The alternate behavior possible when ECHOK or ECHOE are specified with ICANON is permitted as a compromise depending on what the actual terminal hardware can do. Erasing characters and lines is preferred, but is not always possible.

B.7.1.2.6 Special Control Characters

Permitting VMIN and VTIME to overlap with VEOF and VEOL was a compromise for historical implementations. Only when backwards compatibility of object code is a serious concern to an implementor should an implementation continue this practice. Correct applications that work with the overlap (at the source level) should also work if it is not present, but not the reverse.

4042 **B.7.1.2.7 Baud Rate Values**

4043 There is no additional rationale provided for this subclause.

4044 **B.7.1.3 Baud Rate Functions**

4045 The term *baud* is used historically here, but is not technically correct. This is
 4046 properly “bits per second,” which may not be the same as “baud.” However, the
 4047 term is used because of the historical usage and understanding.

4048 These functions do not take numbers as arguments, but rather symbolic names.
 4049 There are two reasons for this:

- 4050 — Historically, numbers were not used because of the way the rate was stored
 4051 in the data structure. This is retained even though an interface function is
 4052 now used.
- 4053 — More importantly, only a limited set of possible rates is at all portable, and
 4054 this constrains the application to that set.

4055 There is nothing to prevent an implementation to accept, as an extension, a
 4056 number (such as 126) if it wished, and because the encoding of the Bxxx symbols
 4057 is not specified, this can be done so no ambiguity is introduced.

4058 Setting the input baud rate to zero was a mechanism to allow for split baud rates.
 4059 Clarifications to this version of POSIX.1 have made it possible to determine if split
 4060 rates are supported and to support them without having to treat zero as a special
 4061 case. Since this functionality is also confusing, it has been declared obsolescent.
 4062 The 0 argument referred to is the literal constant 0, not the symbolic constant B0.
 4063 POSIX.1 does not preclude B0 from being defined as the value 0; in fact, imple-
 4064 mentations will likely benefit from the two being equivalent. POSIX.1 does not
 4065 fully specify whether the previous *cfsetispeed()* value is retained after a *tcgetattr()*
 4066 as the actual value or as zero. Therefore, portable applications should always set
 4067 both the input speed and output speed when setting either.

4068 In historical implementations, the baud rate information is traditionally kept in
 4069 *c_cflag*. Applications should be written to presume that this might be the case
 4070 (and thus not blindly copy *c_cflag*) but not to rely on it, in case it is in some other
 4071 field of the structure. Setting the *c_cflag* field absolutely after setting a baud rate
 4072 is a nonportable action because of this. In general, the unused parts of the flag
 4073 fields might be used by the implementation and should not be blindly copied from
 4074 the descriptions of one terminal device to another.

4075 **B.7.2 General Terminal Interface Control Functions**

4076 The restrictions described in this subclause on access from processes in back-
 4077 ground process groups controls apply only to a process that is accessing its con-
 4078 trolling terminal. (See B.7.1.1.4).

4079 Care must be taken when changing the terminal attributes. Applications should
 4080 always do a *tcgetattr()*, save the *termios* structure values returned, and then do a
 4081 *tcsetattr()* changing only the necessary fields. The application should use the
 4082 values saved from the *tcgetattr()* to reset the terminal state whenever it is done

with the terminal. This is necessary because terminal attributes apply to the underlying port and not to each individual open instance; that is, all processes that have used the terminal see the latest attribute changes.

A program that uses these functions should be written to catch all signals and take other appropriate actions to assure that when the program terminates, whether planned or not, the terminal device's state is restored to its original state. See also B.7.1.

Existing practice dealing with error returns when only part of a request can be honored is based on calls to the *ioctl()* function. In historical BSD and System V implementations, the corresponding *ioctl()* returns zero if the requested actions were semantically correct, even if some of the requested changes could not be made. Many existing applications assume this behavior and would no longer work correctly if the return value were changed from zero to -1 in this case.

Note that either specification has a problem. When zero is returned, it implies everything succeeded even if some of the changes were not made. When -1 is returned, it implies everything failed even though some of the changes were made.

Applications that need all of the requested changes made to work properly should follow *tcsetattr()* with a call to *tcgetattr()* and compare the appropriate field values.

B.7.2.1 Get and Set State

The *tcsetattr()* function can be interrupted in the following situations:

- It is interrupted while waiting for output to drain.
- It is called from a process in a background process group and SIGTTOU is caught.

B.7.2.2 Line Control Functions

There is no additional rationale provided for this subclause.

B.7.2.3 Get Foreground Process Group ID

The *tcgetpgrp()* function has identical functionality to the 4.2BSD *ioctl()* function TIOCGPGRP except for the additional security restriction that the referenced terminal must be the controlling terminal for the calling process.

In the case where there is no foreground process group, returning an error rather than a positive value was considered. This was rejected because existing applications based on either IEEE Std 1003.1-1988 or 4.3BSD are likely to consider errors from this call or the BSD equivalent to be catastrophic and respond inappropriately. Such applications implicitly assume that this case does not exist, and the positive return value is the only solution that permits them to behave properly even when they do encounter it. No application has been identified that can benefit from distinguishing between this case and the case of a valid foreground process group other than its own. Therefore, requiring or permitting any other

4123 solution would cause more application portability problems with no corresponding
 4124 benefit to applications. The value must be positive, not zero, because applications
 4125 may use the negation as the *pid* argument to *kill()*. In addition, the value 1 must
 4126 not be used so that an attempt to send a signal to this (nonexistent) process group
 4127 does not result in broadcasting a signal unintentionally. See also B.7.1.1.2.

4128 B.7.2.4 Set Foreground Process Group ID

4129 The *tcsetpgrp()* function has identical functionality to the 4.2BSD *ioctl()* function
 4130 TIOCSGRP except for the additional security restrictions that the referenced ter-
 4131 minal must be the controlling terminal for the calling process and the specified
 4132 new process group must be currently in use in the caller's session.

4133 B.8 Language-Specific Services for the C Programming Language

4134 See the discussion of C functions in B.1.1.1.

4135 Common usage may be defined by historical publications such as *The C Program-*
 4136 *ming Language* (B46).

4137 The null set of supported languages is allowed.

4138 The list of functions comprises the list of “common-usage” functions and also
 4139 includes those that are not in common usage that are addressed by POSIX.1. The
 4140 rules for common-usage conformance to POSIX.1 address whether the functions
 4141 that are not generally considered in common usage are implemented. There are a
 4142 large number of functions found in various systems that, although frequently
 4143 found, are not broadly enough available to be considered in common usage. The
 4144 *signal()* function (although in common usage) is omitted because applications con-
 4145 forming to POSIX.1 should use the more reliable *sigaction()* interface instead.

4146 B.8.1 Referenced C Language Routines

4147 B.8.1.1 Extensions to Time Functions

4148 System V uses the TZ environment variable to set some information about time.
 4149 It has the form (spaces inserted for clarity):

4150 *std offset dst*

4151 where the first three characters (*std*) are the name of the standard time zone, the
 4152 digits that follow (*offset*) represent the time added to the local time zone to arrive
 4153 at Coordinated Universal Time, and the next three characters (*dst*) are the name
 4154 of the summer time zone. The meaning of *offset* implies that most sites west of
 4155 the Prime Meridian will have a positive offset (preceded by an optional plus sign,
 4156 “+”), while most sites east of the Prime Meridian will have a negative offset (pre-
 4157 ceded by a minus sign, “-”). Both *std* and *offset* are required; if *dst* is missing,
 4158 summer time does not apply.

Currently, the UNIX system *localtime()* function translates a number of seconds since the Epoch into a detailed breakdown of that time. This breakdown includes

- (1) Time of day: Hours, minutes, and seconds.
- (2) Day of the month, month of the year, and the year.
- (3) Day of the week and day of the year (Julian day).
- (4) Whether or not summer (daylight saving) time is in effect.

It is the first and last items that present a problem: the time of the day depends on whether or not summer time is in effect. Whether or not summer time is in effect depends on the locale and date.

Most historical systems had time-zone rules compiled into the C library. These rules usually represented United States rules for 1970 to 1986. This did not accommodate the changes of 1987, nor other world variations ($\frac{1}{2}$ -hour time, double daylight time, and solar time being common, but not complete, examples). Some recent systems addressed these problems in various ways.

Having the rules compiled into the program made binary distributions that accommodated all the variations (including sudden changes to the law), and per-process rule changes, difficult at best.

POSIX.1 includes a way of specifying the time zone in the TZ string, but only permits one time-zone pattern at a time, thus not dealing with different patterns in previous years and with such issues as solar time. Methods exist to deal with all the problems above. The method in POSIX.1 appears to be simpler to implement and may be faster in execution when it is adequate. POSIX.1 also permits an implementation-defined rule set that begins with a colon. (The previous format cannot begin with a colon.)

Rules of the form AAAn or AAAnBBB (the style used in many historical implementations) do not carry with them any statement about the start and end of daylight time (neither the date nor the time of day; the default to 02:00 not applying if no rule is present at all), thus implying that the implementation must provide the appropriate rules. An implementation may provide those rules in any way it sees fit, as long as the constraints implied by the TZ string as provided by the user are met. Specifically, the implementation may use the string as an index into a table, which may reside either on disk or in memory. Such tables could contain rules that are sensitive to the year to which they are applied, again since the user did not specify the exact rule. (Although impractical, every possible TZ string could be represented in a table, as a detail of implementation; the less specific the user is about the TZ string, the more freedom the implementation has to interpret it.)

There is at least one public-domain time-zone implementation (the Olson/Harris method) that uses nonspecific TZ strings and a table, as described previously, and handles all the general time-zone problems mentioned above. This implementation also appears in a late release of 4.3BSD. If this implementation honors all the specifications provided in the TZ string, it would conform to POSIX.1. Nothing precludes the implementation from adding information beyond that given by the user in the TZ string.

The fully specified TZ environment variable extends the historical meaning to also include a rule for when to use standard time and when to use summer time.

4204 Southern hemisphere time zones are supported by allowing the first *rule date*
4205 (change to summer time) to be later in the year than the second *rule date* (change
4206 to standard time).

4207 This mechanism accommodates the “floating day” rules (for example “last Sunday
4208 in October”) used in the United States and Canada (and the European Economic
4209 Community for the last several years). In theory, TZ only has to be set once and
4210 then never touched again unless the law is changed.

4211 Julian dates are proposed with two syntaxes, one zero-based, the other one-based.
4212 They are here for historical reasons. The one-based counting (*J*) is used more
4213 commonly in Europe (and on calendars people may use for reference). The zero-
4214 based counting (*n*) is used currently in some implementations and should be kept
4215 for historical reasons as well as being the only way to specify Leap Day.

4216 It is expected that the leading colon format will allow systems to implement an
4217 even broader range of specifications for the time zone without having to resort to a
4218 file or permit naming an explicit file containing the appropriate rules.

4219 The specification in POSIX.1 for TZ assumes that very few programs need to be
4220 historically accurate as long as the relative timing of two events is preserved.

4221 Summer time is governed by both locale and date. This proposal only handles the
4222 locale dependency. Using an implementation-defined file format for either the
4223 entire TZ variable or to specify the *rules* for a particular time zone is allowed as a
4224 means by which both the locale and date dependency can be handled.

4225 Since historical implementations do not examine TZ beyond the assumed end of
4226 *dst*, it is possible literally to extend TZ and break very little existing software.
4227 Since much historical software does not function outside the US time zones, minor
4228 changes to TZ (such as extending *offset* to be *hh:mm*—as long as the colon and
4229 minutes, *mm*, are optional) should have little effect.

4230 POSIX.1 is intentionally silent about values of TZ that do not fit either of the
4231 specified forms. It simply requires that TZ values that follow those forms be
4232 interpreted as specified.

4233 B.8.1.2 Extensions to *setlocale()* Function

4234 The C Standard [2] defines a collection of interfaces to support internationaliza-
4235 tion. One of the most significant aspects of these interfaces is a facility to set and
4236 query the *international environment*. The international environment is a reposi-
4237 tory of information that affects the behavior of certain functionality, namely

- 4238 (1) Character Handling
- 4239 (2) String Handling (i.e., collating)
- 4240 (3) Date/Time Formatting
- 4241 (4) Numeric Editing

4242 The *setlocale()* function provides the application developer with the ability to set
4243 all or portions, called *categories*, of the international environment. These
4244 categories correspond to the areas of functionality, mentioned above. The syntax
4245 for *setlocale()* is the following:

6 char *setlocale(int *category*, const char **locale*);

7 where *category* is the name of one of five categories, namely

8 LC_CTYPE
9 LC_COLLATE
10 LC_TIME
11 LC_MONETARY
12 LC_NUMERIC

13 In addition, a special value, called LC_ALL, directs *setlocale()* to set all categories.

14 The *locale* argument is a character string that points to a specific setting for the
15 international environment, or locale. There are three preset values for the locale
16 argument, namely

17 "C"	Specifies the minimal environment for C translation. If <i>setlocale()</i> 18 is not invoked, the "C" locale is the default.
19 "POSIX"	Specifies a locale that is the same as "C" for the attributes defined 20 by the C Standard [2] and POSIX.1, but may contain extensions. 21 The wording permits extensions by standards, specifically that of 22 ISO/IEC 9945-2 [B36], which is expected to use the same symbol, 23 and by future versions of POSIX.1.
24 ""	Specifies an implementation-defined native environment.
25 NULL	Used to direct <i>setlocale()</i> to query the current international 26 environment and return the name of the locale.

27 This subclause describes the behavior of an implementation of *setlocale()* and its
28 use of environment variables in controlling this behavior on POSIX.1-based sys-
29 tems. There are two primary uses of *setlocale()*:

- 30 (1) Querying the international environment to find out what it is set to;
- 31 (2) Setting the international environment, or *locale*, to a specific value.

32 The following subclauses describe the behavior of *setlocale()* in these two areas.
33 Since it is difficult to describe the behavior in words, examples will be used to
34 illustrate the behavior of specific uses.

35 To query the international environment, *setlocale()* is invoked with a specific
36 category and the NULL pointer as the locale. The NULL pointer is a special direc-
37 tive to *setlocale()* that tells it to query rather than set the international environ-
38 ment. The following syntax is used to query the name of the international
39 environment:

$$\text{setlocale}\left(\left\{\begin{array}{l} LC_ALL \\ LC_CTYPE \\ LC_COLLATE \\ LC_TIME \\ LC_NUMERIC \\ LC_MONETARY \end{array}\right\}, (\text{char} *) \text{NULL}\right);$$

30 The *setlocale()* function returns the string corresponding to the current interna-
31 tional environment. This value may be used by a subsequent call to *setlocale()* to

4282 reset the international environment to this value. However, it should be noted
 4283 that the return value from *setlocale()* is a pointer to a static area within the func-
 4284 tion and is not guaranteed to remain unchanged [i.e., it may be modified by a sub-
 4285 sequent call to *setlocale()*]. Therefore, if the purpose of calling *setlocale()* is to
 4286 save the value of the current international environment so it can be changed and
 4287 reset later, the return value should be copied to an array of *char* in the calling
 4288 program.

4289 There are three ways to set the international environment with *setlocale()*:

4290 *setlocale(category, string)*

4291 This usage will set a specific *category* in the international
 4292 environment to a specific value corresponding to the value of
 4293 the *string*. A specific example is provided below:

4294 *setlocale(LC_ALL, "Fr_FR.8859");*

4295 In this example, all categories of the international environment
 4296 will be set to the locale corresponding to the string
 4297 "Fr_FR.8859", or to the French language as spoken in France
 4298 using the ISO 8859-1 code set.

4299 If the string does not correspond to a valid locale, *setlocale()*
 4300 will return a NULL pointer and the international environment
 4301 is not changed. Otherwise, *setlocale()* will return the name of
 4302 the locale just set.

4303 *setlocale(category, "C")*

4304 The C Standard [2] states that one locale must exist on all con-
 4305 forming implementations. The name of the locale is "C" and
 4306 corresponds to a minimal international environment needed to
 4307 support the C programming language.

4308 *setlocale(category, "")*

4309 This will set a specific category to an implementation-defined
 4310 default. For POSIX.1-based systems, this corresponds to the
 4311 value of the environment variables.

4312 B.8.2 C Language Input/Output Functions

4313 B.8.2.1 Map a Stream Pointer to a File Descriptor

4314 Without some specification of which file descriptors are associated with these
 4315 streams, it is impossible for an application to set up the streams for another appli-
 4316 cation it starts with *fork()* and *exec*. In particular, it would not be possible to
 4317 write a portable version of the *sh* command interpreter (although there may be
 4318 other constraints that would prevent that portability).

4319 B.8.2.2 Open a Stream on a File Descriptor

4320 The file descriptor may have been obtained from *open()*, *creat()*, *pipe()*, *dup()*, or
 4321 *fcntl()*; inherited through *fork()* or *exec*; or perhaps obtained by implementation-
 4322 dependent means, such as the 4.3BSD *socket()* call.

The meanings of the *type* arguments of *fdopen()* and *fopen()* differ. With *fdopen()*, open for write ("w" or "w+") does not truncate, and append ("a" or "a+") cannot create for writing. There is no need for "b" in the format due to the equivalence of binary and text files in POSIX.1. See B.1.1.1. Although not explicitly required by POSIX.1, a good implementation of append ("a") mode would cause the O_APPEND flag to be set.

B.8.2.3 Interactions of Other *FILE*-Type C Functions

Note that the existence of open streams on a file implies open file descriptors and thus affects the timestamps of the file. The intent is that using *stdio* routines to read a file must eventually update the access time, and using them to write a file must eventually update the modify and change times. However, the exact timing of marking the *st_atime*, *st_ctime*, and *st_mtime* fields cannot be specified, as that would imply a particular buffering strategy.

The purpose of the rules about handles is to allow the writing of a program that uses *stdio* and does some shell-like things; in particular, creating an open file for a child process to use, where both the parent and child wish to use *stdio*, with the consequences of buffering. In most cases, this cannot happen in the C Standard {2} (because there is no way to create a second handle), but the *system()* function can cause this to occur, at least in most historical implementations.

Presently, POSIX.1 deals mostly with output streams; input is implementation defined. It should be possible to make input on seekable devices work for seekable files without affecting buffering strategies significantly. However, the details have not been worked out fully and will be addressed in a future revision of POSIX.1. The requirements on applications are unlikely to change [basically, serving notice to the implementation that the use of a particular handle is (temporarily) completed] and are symmetric to those for output.

There are some implied rules about interprocess synchronization, but no mechanism is given, intentionally. In the simplest case, if the parent meets the requirements on all its files and then performs a *fork()* and a *wait()* before further activity on them [and a *fflush()* on input files after that], the desired synchronization will be achieved. Synchronization could in theory be done with signals, but the only likely case is the one just described. The terms *handle* and *active handle* were required to make the text readable and are not intended for use outside this discussion.

Note that since *exit()* implies *_exit()*, a file descriptor is also closed by *exit()*.

Because a handle is either freshly opened, or if not must have handed off control of the open file description as specified, the new handle is always ready to be used (except for seeks) with no initialization. [A freshly opened stream has not yet done any reads, as required by the C Standard {2}, at least implicitly by the rules associated with *setvbuf()*.]

In requiring the seek to an appropriate location for the new handle, the application is required to know what it is doing if it is passing streams with seeks involved. If the required seek is not done, the results are undefined (and in fact the program probably will not work on many common implementations).

4367 A naive program used as a utility can be reasonably expected to work properly
4368 when the constraints are met by the calling program because it will not hand off
4369 file descriptors except with closes.

4370 The *exec* functions are treated specially because the application should always
4371 *fflush()* everything before performing one of the *exec* functions. If *stdout* is avail-
4372 able on the same open file description after the *exec*, it is a different stream, at
4373 least because any unflushed data will be discarded during the *exec* (similarly for
4374 *stdin*). Process termination is also special because a process terminating due to a
4375 signal or *_exit()* will not have the buffers flushed.

4376 The *fork()* function also must be specially treated because it clones a number of
4377 file descriptors simultaneously. Thus, all of them should be prepared for handoff
4378 before the *fork()*. In effect, *fork()* creates a pair of handles that are improperly
4379 dealt with unless, before the *fork()*, the first part of a handoff occurred. Note that
4380 *fflush(NULL)* in the C Standard [2] is an appropriate way to do this for output. A
4381 subsequent *exec* call [that does not succeed in calling *exit()* in some way] will
4382 reduce the number of handles back to the original value (allowing for files that
4383 are not close-on-exec), and, thus, preparations for *exec* need not necessarily do the
4384 flush. However, because *exit()* closes all streams, if the *exec* fails, the application
4385 must be careful to terminate with *_exit()*.

4386 POSIX.1 does not specify asynchronous I/O, and when dealing with asynchronous
4387 I/O the problem of coordinating access to streams will be more difficult. If asyn-
4388 chronous I/O is provided as an extension, the problems it introduces in this area
4389 should be addressed as part of that extension.

4390 It may be that functions such as *system()* and *popen()*, currently being considered
4391 for ISO/IEC 9945-2 [B36], will have to perform some of these operations.

4392 The introduction of underlying functions allows generic reference to *errno* values
4393 returned by those functions and also to other side effects (as required in the *han-*
4394 *dles* discussion above). It is not intended to specify implementation, although
4395 many implementations may in fact use those functions. The C Standard [2] says
4396 very little about *errno* in the context of *stdio*. In the more restricted POSIX.1
4397 environment, providing a reasonable set of *errno* values become possible.

4398

4399 **B.8.2.3.1 *fopen()***

4400 There is no additional rationale provided for this subclause.

4401 **B.8.2.3.2 *fclose()***

4402 The *fclose()* function is required to synchronize the buffer pointer with the file
4403 pointer (unless it already is, which would be the case at EOF). Functionality
4404 equivalent to

4405 `fseek(stream, ftell(stream), SEEK_SET)`

4406 does this nicely. The exception for devices incapable of seeking is an obvious
4407 requirement, but the implication is that there is no way to reliably read a buffered
4408 pipe and hand off handles. This is the situation in historical implementations
4409 and is inherent in any “read-ahead” buffering scheme. This limitation is also
4410 reflected in the handle hand-off rules.

Note that the last byte read from a stream and the last byte read from an open file description are not necessarily the same; in most cases the open file description's pointer will be past that of the stream because of the stream's read-ahead.

B.8.2.3.3 *freopen()*

There is no additional rationale provided for this subclause.

B.8.2.3.4 *fflush()*

There is no additional rationale provided for this subclause.

B.8.2.3.5 *fgetc()*, *fgets()*, *fread()*, *getc()*, *getchar()*, *gets()*, *scanf()*, *fscanf()*

There is no additional rationale provided for this subclause.

B.8.2.3.6 *fputc()*, *fputs()*, *fwrite()*, *putc()*, *putchar()*, *puts()*, *printf()*, *vprintf()*, *fprintf()*

There is no additional rationale provided for this subclause.

B.8.2.3.7 *fseek()*, *rewind()*

The *fseek()* function must operate as specified to make the case where seeking is being done work. The key requirement is to avoid an optimization such that an *fseek()* would not result in an *lseek()* if the *fseek()* pointed within the current buffer. This optimization is valuable in general, so it is only required after an *fflush()*.

B.8.2.3.8 *perror()*

There is no additional rationale provided for this subclause.

B.8.2.3.9 *tmpfile()*

There is no additional rationale provided for this subclause.

B.8.2.3.10 *ftell()*

In append mode, a *fflush()* will change the seek pointer because of possible writes by other processes on the same file. An *fseek()* reflects the underlying file's file offset, which is not necessarily the end of the file. Implementors should be aware that the operating system itself (not some in-memory approximation) of the file offset should be queried when in append mode.

B.8.2.3.11 Error Reporting

POSIX.1 intentionally does not require that all errors detected by the underlying functions be detected by the functions listed here. There are many reasonable cases where this might not occur; for example, many of the functions with *write()* as an underlying function might not detect a number of error conditions in cases where they simply buffer output for a subsequent flush.

4446 [ENOMEM] was considered for addition as an explicit possible error because most
 4447 implementations use *malloc()*. This was not done because the scope does not
 4448 include “out of resource” errors. Nevertheless this is the most likely error to be
 4449 added to the possible error conditions. Other implementation-defined errors, par-
 4450 ticularly in the *f*open()* family, are to be expected, and the generic rules about
 4451 adding (or deleting) possible errors apply, except that it is expected that
 4452 implementation-defined changes in the error set returned by *open()* would also
 4453 apply to *fopen()* [unless the condition cannot possibly happen in *fopen()*, which
 4454 may be possible, but appears unlikely].

4455 **B.8.2.3.12 *exit()*, *abort()***

4456 POSIX.1 intends that processing related to the *abort()* function will occur unless
 4457 “the signal SIGABRT is being caught, and the signal handler does not return,” as
 4458 defined by the C Standard [2]. This processing includes at least the effect of
 4459 *fclose()* on all open streams, and the default actions defined for SIGABRT.

4460 The *abort()* function will override blocking or ignoring the SIGABRT signal.
 4461 Catching the signal is intended to provide the application writer with a portable
 4462 means to abort processing, free from possible interference from any
 4463 implementation-provided library functions.

4464 Note that the term “program termination” in the C Standard [2] is equivalent to
 4465 “process termination” in POSIX.1.

4466 **B.8.2.4 Operations on Files — the *remove()* Function**

4467 There is no additional rationale provided for this subclause.

4468 **B.8.3 Other C Language Functions**

4469 **B.8.3.1 Nonlocal Jumps**

4470 The C Standard [2] specifies various restrictions on the usage of the *setjmp()*
 4471 macro in order to permit implementors to recognize the name in the compiler and
 4472 not implement an actual function. These same restrictions apply to the *sig-*
 4473 *setjmp()* macro.

4474 There are processors that cannot easily support these calls, but this was not con-
 4475 sidered a sufficient reason to exclude them.

4476 The distinction between *setjmp()/longjmp()* and *sigsetjmp()/siglongjmp()* is only
 4477 significant for programs that use the *sigaction()*, *sigprocmask()*, or *sigsuspend()*
 4478 functions. Since earlier implementations did not have signal masks, only a single
 4479 pair was provided.

4480 4.2BSD and 4.3BSD systems provide functions named *_setjmp()* and *_longjmp()*
 4481 that, together with *setjmp()* and *longjmp()*, provide the same functionality as *sig-*
 4482 *setjmp()* and *siglongjmp()*. On those systems, *setjmp()* and *longjmp()* save and
 4483 restore signal masks, while *_setjmp()* and *_longjmp()* do not. On System V
 4484 Release 3 and in corresponding issues of the *SVID* [B39], *setjmp()* and *longjmp()*
 4485 are explicitly defined not to save and restore signal masks. In order to permit

existing practice in both cases, the relation of *setjmp()* and *longjmp()* to signal masks is not specified, and a new set of functions is defined instead.

B.8.3.2 Set Time Zone

There is no additional rationale provided for this subclause.

B.8.3.3 Omitted Memory Management

The *brk()* and *sbrk()* functions frequently were proposed for inclusion in POSIX.1, but they were excluded deliberately. See also B.1.1. The rationale for including them is usually addressed to the argument that it is the *sbrk()* primitive that makes it possible to implement some more general heap management system, such as that provided for C by *malloc()*. The need for such functionality is fully understood, but specifying it as a part of a standard would have the effect of limiting the number of architectures that could support POSIX.1. It might also constrain languages whose memory-management model was not served by the *sbrk()* model.

Memory management is not excluded from POSIX.1: POSIX.1 relies on the language to provide it, and in the C binding (as reflected in Section 8) it is provided by *malloc()*. It would be provided by *new()* in Pascal. In a language like FORTRAN, which does not supply memory management to the user, it would be undesirable to force the language binding to attempt to include such a function. It is reasonable to imagine a language that required a more powerful primitive than *sbrk()* to be implemented, and standardizing *sbrk()* would only constrain such future languages.

POSIX.1 is silent about mixed languages. Mixing languages that provide incompatible memory-management mechanisms can yield unpredictable results. Future standards that address mixing of languages should consider this issue.

Architectures that could not support *sbrk()* are also a limiting factor. In particular, architectures that do not present a model of a single linear address space would be severely constrained by *sbrk()*, but are not so constrained by *malloc()* or *new()*.

Each language should specify the memory-management primitives best suited to that language. Whether the implementor chooses to use a more primitive mechanism to implement that, or the implementor chooses to directly implement the language function in the kernel, is not a proper concern of the developers of POSIX.1, nor should it be for any portable application. An application that presumes the *sbrk()* model of memory management will not port to all architectures in any case, for the same reasons that *sbrk()* itself does not work on those architectures. No true gain in application portability would be achieved by mandating such an interface. This implies that an implementor of software that wishes to port to multiple platforms and that attempts to implement its own memory management rather than relying on language-supplied functions must be prepared to deal with multiple platform-supplied primitives and, because it is doing its own memory management inherently, cannot be considered, or be made to be, portable in that regard.

4529 **B.9 System Databases**

4530 At one time, this section was entitled Passwords, but this title was changed as all
4531 references to a “password file” were changed to refer to a “user database.”

4532 **B.9.1 System Databases**

4533 There are no references in POSIX.1 to a *passwd* file or a *group* file, and there is no
4534 requirement that the *group* or *passwd* databases be kept in files containing edit-
4535 able text. Many large timesharing systems use *passwd* databases that are
4536 hashed for speed. Certain security classifications prohibit certain information in
4537 the *passwd* database from being publicly readable.

4538 The term “encoded” is used instead of “encrypted” in order to avoid the implemen-
4539 tation connotations (such as reversibility or use of a particular algorithm) of the
4540 latter term.

4541 The *getgrent()*, *setgrent()*, *endgrent()*, *getpwent()*, *setpwent()*, and *endpwent()*
4542 functions are not included in POSIX.1 because they provide a linear database
4543 search capability that is not generally useful [the *getpwuid()*, *getpwnam()*, *get-*
4544 *grgid()*, and *getgrnam()* functions are provided for keyed lookup] and because in
4545 certain distributed systems, especially those with different authentication
4546 domains, it may not be possible or desirable to provide an application with the
4547 ability to browse the system databases indiscriminately.

4548 A change from historical implementations is that the structures used by these
4549 functions have fields of the types *gid_t* and *uid_t*, which are required to be defined
4550 in the header `<sys/types.h>`. POSIX.1 has not changed the synopses of these
4551 functions to require the inclusion of this header, since that would invalidate a
4552 large number of existing applications. Implementations must ensure that these
4553 types are defined by the inclusion of `<grp.h>` and `<pwd.h>`, respectively, without
4554 imposing any namespace pollution or errors from redefinition of types.

4555 POSIX.1 is silent about the content of the strings containing user or group names.
4556 These could be digit strings. POSIX.1 is also silent as to whether such digit
4557 strings bear any relationship to the corresponding (numeric) user or group ID.

4558 **B.9.2 Database Access**4559 **B.9.2.1 Group Database Access**

4560 There is no additional rationale provided for this subclause.

4561 **B.9.2.2 User Database Access**

4562 There is no additional rationale provided for this subclause.

B.10 Data Interchange Format

B.10.1 Archive/Interchange File Format

There are three areas of interest associated with file interchange:

- (1) *Media*. There are other existing standards that define the media used for data interchange.
- (2) *User Interface*. This rightfully should be in the shell and utilities standard, under development as ISO/IEC 9945-2 [B36].
- (3) *Format of the Data*. None of the groups currently developing POSIX standards address topics that match this area. The groups felt that this area is closest to the types of things that should be in the POSIX.1 document, as the level of that document most closely matches the level of data required.

There are two programs in wide use today: `tar` and `cpio`. There are many supporters for each program. Four options were considered for POSIX.1:

- (1) Make both formats optional. This was considered unacceptable because it does not allow any portable method for data interchange.
- (2) Require one format.
- (3) Require one format with the other optional.
- (4) Require both formats.

Both the Extended `cpio` and the Extended `tar` Formats are required by POSIX.1.

There are a number of concerns about defining extensions that are known to be required by historical implementations. Failure to specify a consistent method to implement these extensions will limit portability of the data and, more importantly, will create confusion if these extensions are later standardized.

Two of these extensions that should be documented are symbolic links, which were defined by 4.2BSD and 4.3BSD systems, and high-performance (or contiguous) files, which exist in a number of implementations and are now being considered for future amendments to POSIX.1.

By defining these extensions, implementors are able to recognize these features and take appropriate implementation-defined actions for these files. For example, a high-performance file could be converted to a regular file if the system did not support high-performance files; symbolic links might be replaced by normal hard links.

The policy of not defining user interfaces to utilities preempted any description of a `tar` or `cpio` command. The behavior of the former command was described in some detail in previous drafts.

The possibilities for transportable media include, but are not limited to

- (1) 12,7 mm (0,5 in) magnetic tape, 9 track, 63 b/mm (1 600 bpi)
- (2) 12,7 mm (0,5 in) magnetic tape, 9 track, 246 c/mm (6 250 cpi)

4602 (3) QIC-11, 6,30 mm (0,25 in) streamer tape |

4603 (4) QIC-24, 6,30 mm (0,25 in) streamer tape |

4604 (5) 130 mm (5,25 in) diskettes, 9 512-byte sectors/track, 3,8 tpm (96 tpi) |

4605 (6) 130 mm (5,25 in) diskettes, 9 512-byte sectors/track, 1,9 tpm (48 tpi) |

4606 When selecting media, issues such as character frame size also need to be |

4607 addressed. The easiest environment for interchange occurs when 8-bit frames are |

4608 used. |

4609 The utilities are not restricted to work only with *transportable* media: existing |

4610 related utilities are often used to transport data from one place to another in the |

4611 file hierarchy. |

4612 The formats are included to provide implementation-independent ways to move |

4613 files from one system to another and also to provide ways for a user to save data |

4614 on a transportable medium to be restored at a later date. Unfortunately, these |

4615 two goals can contradict each other, as system security problems are easy to find |

4616 in tape systems if they are not protected. Thus, there are strict requirements |

4617 about how the mechanism to copy files shall react when operated by both |

4618 privileged and nonprivileged users. The general concept is that a privileged (his- |

4619 torically using the ISUID bit in the file's mode with the owner UID of the file set to |

4620 super-user) version of the utility (or one operated by a privileged user) can be |

4621 used as a save/restore scheme, but a nonprivileged version is used to interpret |

4622 media from a different system without compromising system security. |

4623 Regardless of the archive format used, guidelines should be observed when writ- |

4624 ing tapes to be read on other systems. Assuming the target system conforms to |

4625 POSIX.1, archives created should only use definitions found in POSIX.1 (e.g., file |

4626 types, minimum values as found in Section 2) and should only use relative path- |

4627 names (i.e., no leading slash). |

4628 Both *tar* and *cpio* formats have traditionally been used for both exchange of |

4629 information and archiving. These formats have a number of features that facili- |

4630 tate archiving, for example, the ability to store information about a file that is a |

4631 device. POSIX.1 does not assume this kind of data is portable. It is intended that |

4632 these formats provide for the portable exchange of source information between |

4633 dissimilar systems. This requires specification of the character set to be used |

4634 (ISO/IEC 646 {1}) when these formats are used to write source information. The |

4635 1990 version of ISO/IEC 646 {1} IRV was selected as the international character set |

4636 that corresponds most directly to the ASCII set used in many historical implemen- |

4637 tations. The 1990 version was chosen over the 1983 version because it defines |

4638 '\$' as the currency symbol in the IRV, as opposed to the starburst-like generic |

4639 currency symbol. Note that ISO/IEC 646 {1} is a safe lowest-common-denominator |

4640 character set and that interchange of larger character sets is permitted by mutual |

4641 agreement. Using any other character set (such as ISO 8859-1 {B34} or some mul- |

4642 ti-byte character set) reduces the number of machines to which interchange is |

4643 guaranteed. |

4644 All data written by format-creating utilities and read by format-reading utilities |

4645 is an ordered stream of bytes. The first byte of the stream should be first on the |

4646 medium, the second byte second, etc. On systems where the hardware swaps |

4647 bytes or otherwise rearranges the byte stream on output or input, the

implementor of these utilities must compensate for this so that the data on the storage device retains its ordered nature.

POSIX.1 describes two different formats for data archiving and interchange. Strong support for both formats was evident through the balloting process. This is a clear indication of the need for both formats due to existing practice. The balloting process also defined a number of deficiencies of each format. The strong support indicates that these deficiencies are not sufficient to remove either format from POSIX.1, but will need to be addressed in future amendments to POSIX.1. It was not practical to remedy these deficiencies during the balloting process. Considerable thought and review must occur before making any changes to these formats. It was felt that the best solution is to advise implementors and application writers of these deficiencies by documenting them in the rationale and to include both formats in POSIX.1.

The developers of POSIX.1 recognize the desirability for migration toward one common format and have been made aware of some strong inputs to consider both formats in light of existing practice, current technology trends, and the POSIX standards activities such as security and high-performance systems, and to develop one format that is technically superior. This format will be incorporated into a future amendment to POSIX.1 when it is developed.

The deficiencies that have been identified in the existing formats are as follows. The size of a file link is limited to 100 characters in `tar`. A number of fields in the `cpio` header (`c_filesize`, `c_dev`, `c_ino`, `c_mode`, and `c_rdev`) are too short to support values that POSIX.1 allows these fields to contain. Some existing implementations and current trends in development will require the ability to represent even larger values in these fields. The `cpio` format does not provide a mechanism to represent the user and group IDs symbolically, and a range of implementation-defined file types have not been reserved for the user. The `cpio` format specification does not reserve any formats for implementation-defined usage. The extensions that have been made to `cpio` for POSIX.1 are compatible with existing versions of `cpio`. Correction of some of these deficiencies would make existing versions of `cpio` behave unpredictably. When these changes are made the `cpio` magic number will have to be changed.

This clause uses the term *file name*; note that *filename* and *file name* are not synonyms; the latter is a synonym for *pathname*, in that it includes the slashes between filenames.

In earlier drafts, the word “local” was used in the context of “file system” and was taken (incorrectly) to be related to “remotely mounted file system.” This was not intended. The term “(local) file system” refers to the file hierarchy as seen by the utilities, and “local” was removed because of this confusion.

B.10.1.1 Extended `tar` Format

The original model for this facility is the 4.3BSD or Version 7 `tar` program and format, but the format given here is an extension of the traditional `tar` format. The name USTAR was adopted to reflect this.

This description reflects numerous enhancements over previous versions. The goal of these changes was not only to provide the functional enhancements

4693 desired, but also to retain compatibility between new and old versions. This com-
4694 patibility has been retained. Archives written using the old archive format are
4695 compatible with the new format. Archives written using this new format may be
4696 read by applications designed to use the old format as long as the functional
4697 enhancements provided here are not used. This means the user is limited to
4698 archiving only regular type files and nonsymbolic links to such files.

4699 Implementors should be aware that the previous file format did not include a
4700 mechanism to archive directory type files. For this reason, the convention of
4701 using a file name ending with slash was adopted to specify a directory on the
4702 archive.

4703 The total size of the name and prefix fields have been set to meet the minimum
4704 requirements for (PATH_MAX). If a pathname will fit within the name field, it is
4705 recommended that the pathname be stored there without the use of the prefix
4706 field. Although the name field is known to be too small to contain (PATH_MAX)
4707 characters, the value was not changed in this version of the archive file format to
4708 retain backward compatibility, and instead the *prefix* was introduced. Also,
4709 because of the earlier version of the format, there is no way to remove the restric-
4710 tion on the *linkname* field being limited in size to just that of the *name* field.

4711 The *size* field is required to be meaningful in all implementation extensions,
4712 although it could be zero. This is required so that the data blocks can always be
4713 properly counted.

4714 It is suggested that if device special files need to be represented that cannot be
4715 represented in the standard format that one of the extension types ('A'-'Z') be
4716 used, and that the additional information for the special file be represented as
4717 data and be reflected in the size field.

4718 Attempting to restore a special file type, where it is converted to ordinary data
4719 and conflicts with an existing file name, need not be specially detected by the util-
4720 ity. If run as an ordinary user, a format-reading utility should not be able to
4721 overwrite the entries in, for example, /dev in any case (whether the file is con-
4722 verted to another type or not). If run as a privileged user, it should be able to do
4723 so, and it would be considered a bug if it did not. The same is true of ordinary
4724 data files and similarly named special files; it is impossible to anticipate the
4725 user's needs (who could really intend to overwrite the file), so the behavior should
4726 be predictable (and thus regular) and rely on the protection system as required.

4727 The values '2' and '7' in the typeflag field are intended to define how symbolic
4728 links and contiguous files can be stored in a tar archive. POSIX.1 does not
4729 require the symbolic link or contiguous file extensions, but does define a standard
4730 way of archiving these files so that all conforming systems can interpret these file
4731 types in a meaningful and consistent manner. On a system that does not support
4732 extended file types, the format-interpreting utility should do the best it can with
4733 the file and go on to the next.

4734 B.10.1.2 Extended cpio Format

4735 The model for this format is the existing System V `cpio -c` data interchange for-
4736 mat. This model documents the portable version of `cpio` format and not the
4737 binary version. It has the flexibility to transfer data of any type described within

the POSIX.1 standard, yet is extensible to transfer data types specific to extensions beyond POSIX.1 (e.g., symbolic links or contiguous files). Because it describes existing practice, there is no question of maintaining upward compatibility.

This subclause does not standardize behavior for the utility when the file type is not understood or supported. It is useful for the utility to report to the user whatever action is taken in this case, though POSIX.1 neither requires nor recommends this.

B.10.1.2.1 *cpio* Header

There has been some concern that the size of the *c_ino* field of the header is too small to handle those systems that have very large i-node numbers. However, the *c_ino* field in the header is used strictly as a hard link resolution mechanism for archives. It is not necessarily the same value as the i-node number of the file in the location from which that file is extracted.

B.10.1.2.2 *cpio* File Name

For most historical implementations of the *cpio* utility, {PATH_MAX} bytes can be used to describe the pathname without the addition of any other header fields (the null byte would be included in this count). {PATH_MAX} is the minimum value for pathname size, documented as 256 bytes in Section 2. However, an implementation may use *c_namesize* to determine the exact length of the pathname. With the current description of the *cpio* header, this pathname size can be as large as a number that is described in six octal digits.

B.10.1.2.3 *cpio* File Data

There is no additional rationale provided for this subclause.

B.10.1.2.4 *cpio* Special Entries

These are provided to maintain backward compatibility.

B.10.1.2.5 *cpio* Values

Three values are documented under the *c_mode* field values to provide for extensibility for known file types:

0110 000	Reserved for contiguous files. The implementation may treat the rest of the information for this archive like a regular file. If this file type is undefined, the implementation may create the file as a regular file.
0120 000	Reserved for files with symbolic links. The implementation may store the link name within the data portion of the file. If this type is undefined, the implementation may not know how to link this file or be able to understand the data section. The implementation may decide to ignore this file type and output a warning message.

4777 0140000 Reserved for sockets. If this type is undefined on the target
4778 system, the implementation may decide to ignore this file type
4779 and output a warning message.

4780 This provides for extensibility of the `cpio` format while allowing for the ability to
4781 read old archives. Files of an unknown type may be read as “regular files” on
4782 some implementations. On a system that does not support extended file types,
4783 the format-interpreting utility should do the best it can with the file and go on to
4784 the next.

4785 **B.10.1.3 Multiple Volumes**

4786 Multivolume archives have been introduced in a manner that has become a *de*
4787 *facto* standard in many implementations. Though it is not required by POSIX.1,
4788 classical implementations of the format-reading and -creating utility, upon read-
4789 ing logical end-of-file, check to see if an error channel is open to a controlling ter-
4790 minal. The utility then produces a message requesting a new medium to be made
4791 available. The utility waits for a new medium to be made available by attempting
4792 to read a message to restart from the controlling terminal. In all cases, the com-
4793 munication with the controlling terminal is in an implementation-defined
4794 manner.

4795 This subclause (10.1.3) is intended to handle the issue of multiple volume
4796 archives. Since the end-of-medium and transition between media are not properly
4797 part of POSIX.1, the transition is described in terms of files; the word “file” is used
4798 in a very broad, but correct, sense—a tape drive is a file. The intent is that files
4799 will be read serially until the end-of-archive indication is encountered and that
4800 file or media change will be handled by the utilities in an implementation-defined
4801 manner.

4802 Note that there was an issue with the representation of this on magnetic tape,
4803 and POSIX.1 is intended to be interpreted such that each byte of the format is
4804 represented on the media exactly once. In some current implementations, it is
4805 not deterministic whether encountering the end-of-medium reflector foil on mag-
4806 netic tape during a write will yield an error during a subsequent *read()* of that
4807 record, or if that record is actually recorded on the tape. It is also possible that
4808 *read()* will encounter the end-of-medium when end-of-medium was not encoun-
4809 tered when the data was written. This has to do with conditions where the end of
4810 [magnetic] record is in such a position that the reflector foil is on the verge of
4811 being detected by the sensor and is detected during one operation and not on a
4812 later one, or vice versa.

4813 An implementation of the format-creating utility must assure when it writes a
4814 record that the data appears on the tape exactly once. This implies that the pro-
4815 gram and the tape driver work in concert. An implementation of the format-
4816 reading utility must assure that an error in a boundary condition described above
4817 will not cause loss of data.

4818 The general consensus was that the following would be considered as correct
4819 operation of a tape driver when end-of-medium is detected:

4820 (1) During writing, either

- 821 (a) The record where the reflector spot was detected is backspaced over
822 by the driver so that the trailing tape mark that will be written on
823 *close()* will overwrite. Writing the tape mark should not yield an
824 end-of-medium condition, or
- 825 (b) The condition is reported as an error on the *write()* following the
826 one where the end-of-medium is detected (the one where the end-
827 of-medium is actually detected completing successfully). No data
828 will be actually transferred on the *write()* reporting the error. The
829 subsequent *close()* would write a tape mark following the last
830 record actually written. Writing the tape mark, and writing any
831 subsequent records, should not yield any end-of-medium conditions.
- 832 [The latter behavior permits the implementation of ANSI standard labels
833 because several records (the trailer records) can be written after the end-
834 of-medium indications. It also permits dealing with, for example, COBOL
835 "ON" statements.]
- 836 (2) During reading, the end-of-medium indicator is simply ignored, presum-
837 ing that a tape mark (end-of-file) will be recorded on the magnetic
838 medium and that the reflector foil was advisory only to the *write()*.
- 839 Systems where these conditions are not met by the tape driver should assure that
840 the format-creating and -reading utilities assure proper representation and
841 interpretations of the files on the media in a way consistent with the above recom-
842 mendations.
- 843 The typical failures on systems that do not meet the above conditions are either
- 844 (1) To leave the record written when the end-of-medium is encountered on
845 the tape, but to report that it was not written. The format-creating utility
846 would then rewrite it, and then the format-reading utility could see
847 the record twice if the end-of-medium is not sensed during the read
848 operations, or
- 849 (2) The *write()* occurs uneventfully, but the *read()* senses the error and does
850 not actually see the data, causing a record to be omitted.
- 851 Nothing in POSIX.1 requires that end-of-medium be determined by anything on
852 the medium itself (for example, a predetermined maximum size would be an
853 acceptable solution for the format-creating utility). The format-reading utility
854 must be able to *read()* tapes written by machines that do use the whole medium,
855 however.
- 856 On media where end-of-medium and end-of-file are reliably coincident, such as
857 disks, end-of-medium and end-of-file can be treated as synonyms.
- 858 Note that partial physical records [corresponding to a single *write()*] can be writ-
859 ten on some media, but that only full physical records will actually be written to
860 magnetic tape, given the manner in which the tape operates.

Annex C (informative)

Header Contents Samples

1 The material in this informative annex serves as an index to which symbols
2 should appear in which headers in a system that conforms to POSIX.1 with C
3 Standard Language-Dependent System support.

4 This is only an index, and any conflicts with the actual body of any relevant stan-
5 dard shall be resolved in favor of that standard. The actual body of the declara-
6 tion was omitted in part because this is an index and in part to avoid any possible
7 conflict with the standards.

8 Where it is known that a symbol or header is not required for Common Usage C
9 Language-Dependent System support, the name is followed by an asterisk (*).
10 Omission of an asterisk does not imply that the symbol is required for Common-
11 Usage C. For Common-Usage C, although the location of symbols is typical, it is
12 not to be taken as a requirement: POSIX.1 is quite explicit that there is no
13 requirement except that differences from the C Standard {2} be documented.

14 Generally, where it is stated that functions are defined in a header, macros are
15 permitted as acceptable alternatives by both standards. See the bodies of the
16 standards for details.

17 **<assert.h>**

18 The header defines the macro

19 *assert()*

20 and makes reference to the macro

21 **NDEBUG**

22 **<ctype.h>**

23 The header declares the functions

24	<i>isalnum()</i>	<i>isdigit()</i>	<i>islower()</i>	<i>ispunct()</i>	<i>isupper()</i>	<i>tolower()</i>
25	<i>isalpha()</i>	<i>isgraph()</i>	<i>isprint()</i>	<i>isspace()</i>	<i>isxdigit()</i>	<i>toupper()</i>
26	<i>isctrl()</i>					

<dirent.h>

The header defines the typedef

DIR

and declares the structure

dirent

with structure element member

d_name

and declares functions

closedir() *opendir()* *readdir()* *rewinddir()*

<errno.h>

The header defines the macros

E2BIG	EEXIST	EMLINK	ENOMEM	EPERM
EACCES	EFAULT	ENAMETOOLONG	ENOSPC	EPIPE
EAGAIN	EBIG	ENFILE	ENOSYS	ERANGE
EBADF	EINTR	ENODEV	ENOTDIR	EROFS
EBUSY	EINVAL	ENOENT	ENOTEMPTY	ESPIPE
ECHILD	EIO	ENOEXEC	ENOTTY	ESRCH
EDEADLK	EISDIR	ENOLCK	ENXIO	EXDEV
EDOM	EMFILE			

and declares the external variable

errno

<fcntl.h>

The header defines the macros

FD_CLOEXEC	F_SETFD	O_ACCMODE	O_NONBLOCK
F_DUPFD	F_SETFL	O_APPEND	O_RDONLY
F_GETFD	F_SETLK	O_CREAT	O_RDWR
F_GETFL	F_SETLKW	O_EXCL	O_TRUNC
F_GETLK	F_UNLCK	O_NOCTTY	O_WRONLY
F_RDLCK	F_WRLCK		

and declares the structure

flock

with structure elements

```

59      l_len      l_pid      l_start      l_type      l_whence      |
60  and the functions      |
61      creat()  fcntl()  open()      |
62  and may contain the macros      |
63      SEEK_CUR  S_IROTH  S_IRWXU  S_ISFIFO  S_IWGRP  S_IXGRP      |
64      SEEK_END  S_IRUSR  S_ISBLK  S_ISGID  S_IWOTH  S_IXOTH      |
65      SEEK_SET  S_IRWXG  S_ISCHR  S_ISREG  S_IWUSR  S_IXUSR      |
66      S_IRGRP   S_IRWXO  S_ISDIR  S_ISUID      |
67  <float.h>*      |
68  The header defines the macros      |
69      DBL_DIG*      FLT_EPSILON*      LDBL_DIG*      |
70      DBL_EPSILON*  FLT_MANT_DIG*      LDBL_EPSILON*      |
71      DBL_MANT_DIG*  FLT_MAX*      LDBL_MANT_DIG*      |
72      DBL_MAX*      FLT_MAX_10_EXP*  LDBL_MAX*      |
73      DBL_MAX_10_EXP*  FLT_MAX_EXP*  LDBL_MAX_10_EXP*      |
74      DBL_MAX_EXP*    FLT_MIN*      LDBL_MAX_EXP*      |
75      DBL_MIN*      FLT_MIN_10_EXP*  LDBL_MIN*      |
76      DBL_MIN_10_EXP*  FLT_MIN_EXP*  LDBL_MIN_10_EXP*      |
77      DBL_MIN_EXP*    FLT_RADIX*      LDBL_MIN_EXP*      |
78      FLT_DIG*      FLT_ROUNDS*      |
79  <grp.h>      |
80  The header declares the structure      |
81      group      |
82  with structure elements      |
83      gr_gid      gr_mem      gr_name      |
84  and the functions      |
85      getgrgid()  getgrnam()      |

```

<limits.h>

The header defines the macros

ARG_MAX●	NGROUPS_MAX	USHRT_MAX
CHAR_BIT	OPEN_MAX●	_POSIX_ARG_MAX
CHAR_MAX	PATH_MAX●	_POSIX_CHILD_MAX
CHAR_MIN	PIPE_BUF●	_POSIX_LINK_MAX
CHILD_MAX●	SCHAR_MAX	_POSIX_MAX_CANON
INT_MAX	SCHAR_MIN	_POSIX_MAX_INPUT
INT_MIN	SHRT_MAX	_POSIX_NAME_MAX
LINK_MAX●	SHRT_MIN	_POSIX_NGROUPS_MAX
LONG_MAX	SSIZE_MAX	_POSIX_OPEN_MAX
LONG_MIN	STREAM_MAX□	_POSIX_PATH_MAX
MAX_CANON●	TZNAME_MAX	_POSIX_PIPE_BUF
MAX_INPUT●	UCHAR_MAX	_POSIX_SSIZE_MAX
MB_LEN_MAX	UINT_MAX	_POSIX_STREAM_MAX
NAME_MAX●	ULONG_MAX	_POSIX_TZNAME_MAX

The macros marked with □ shall be omitted from <limits.h> on specific implementations where the corresponding value is greater than or equal to the stated minimum, but is indeterminate. The macros marked with ● shall be omitted from <limits.h> on specific implementations where the corresponding value is greater than or equal to the stated minimum, but where the value can vary depending on the file to which it is applied.

<locale.h>

The header defines the macros

LC_ALL*	LC_CTYPE*	LC_NUMERIC*	NULL*
LC_COLLATE*	LC_MONETARY*	LC_TIME*	

and declares the structure

*lconv**

with structure elements

<i>currency_symbol*</i>	<i>mon_decimal_point*</i>	<i>negative_sign*</i>
<i>decimal_point*</i>	<i>mon_grouping*</i>	<i>p_cs_precedes*</i>
<i>frac_digits*</i>	<i>mon_thousands_sep*</i>	<i>p_sep_by_space*</i>
<i>grouping*</i>	<i>n_cs_precedes*</i>	<i>p_sign_posn*</i>
<i>int_curr_symbol*</i>	<i>n_sep_by_space*</i>	<i>positive_sign*</i>
<i>int_frac_digits*</i>	<i>n_sign_posn*</i>	<i>thousands_sep*</i>

and the functions

localeconv() setlocale()**


```

123  <math.h>
124  The header defines the macro
125      HUGE_VAL
126  and declares the functions
127      acos()    ceil()    exp()    fmod()    log10()    pow()    sqrt()
128      asin()    cos()    fabs()    frexp()    log()    sin()    tan()
129      atan2()    cosh()    floor()    ldexp()    modf()    sinh()    tanh()
130      atan()
131  <pwd.h>
132  The header defines the structure
133      passwd
134  with structure elements
135      pw_dir    pw_gid    pw_name    pw_shell    pw_uid
136  and declares the functions
137      getpwnam()    getpwuid()
138  <setjmp.h>
139  The header defines the types
140      jmp_buf    sigjmp_buf
141  and declares the functions
142      longjmp()    setjmp()    siglongjmp()    sigsetjmp()
143  Note that the C Standard {2} and this part of ISO/IEC 9945 both permit these
144  functions to be defined solely as macros.
145  <signal.h>
146  The header defines the macros
147      SA_NOCLDSTOP    SIGHUP    SIGQUIT    SIGTTIN    SIG_DFL
148      SIGABRT          SIGILL    SIGSEGV    SIGTTOU    SIG_ERR*
149      SIGALRM          SIGINT    SIGSTOP    SIGUSR1    SIG_IGN
150      SIGCHLD          SIGKILL    SIGTERM    SIGUSR2    SIG_SETMASK
151      SIGCONT          SIGPIPE    SIGTSTP    SIG_BLOCK    SIG_UNBLOCK
152      SIGFPE
153  and the types

```

sig_atomic_t* sigset_t

and declares the structure

sigaction

with structure elements

sa_flags *sa_handler* *sa_mask*

and the functions

<i>kill()</i>	<i>sigaddset()</i>	<i>sigfillset()</i>	<i>sigpending()</i>
<i>raise()*</i>	<i>sigdelset()</i>	<i>sigismember()</i>	<i>sigprocmask()</i>
<i>sigaction()</i>	<i>sigemptyset()</i>	<i>signal()*</i>	<i>sigsuspend()</i>

<stdarg.h>*

The header defines the macros

*va_arg** *va_end** *va_list** *va_start**

<stddef.h>*

The header defines the macros

NULL* *offsetof**

and the types

*ptrdiff_t** *size_t** *wchar_t**

<stdio.h>

The header defines the macros

BUFSIZ	L_tmpnam*	STREAM_MAX*	<i>stdout</i>
EOF	NULL	TMP_MAX	_IOFBF*
FILENAME_MAX*	SEEK_CUR	<i>stderr</i>	_IOLBF*
L_ctermid	SEEK_END	<i>stdin</i>	_IONBF*
L_cuserid	SEEK_SET		

NOTE: The `L_userid` symbol is permitted in this header, but need not be supplied. See 2.7.2.

and the types

fpos_t size_t*

and declares the type

```

182      FILE
183  and the functions

184      clearerr()    fileno()    fsetpos()*    putchar()    sprintf()
185      fclose()     fopen()     ftell()     puts()     sscanf()
186      fdopen()     fprintf()    fwrite()    remove()    tmpfile()
187      feof()       fputc()     getc()      rename()    tmpnam()
188      ferror()     fputs()     getchar()   rewind()    ungetc()
189      fflush()     fread()     gets()      scanf()     vfprintf()*
190      fgetc()      freopen()   perror()    setbuf()    vprintf()*
191      fgetpos()*    fscanf()    printf()    setvbuf()*   vsprintf()*
192      fgets()     fseek()     putc()

193  <stdlib.h>
194  The header defines the macros

195      EXIT_FAILURE  MB_CUR_MAX*  RAND_MAX
196      EXIT_SUCCESS  NULL

197  and the types

198      div_t*        ldiv_t*        size_t        wchar_t*

199  and declares the functions

200      abort()       bsearch()       labs()*        qsort()        strtol()*
201      abs()         calloc()        ldiv()*        rand()         strtoul()*
202      atexit()*      div()*           malloc()      realloc()      system()*
203      atof()        exit()          mblen()*       srand()         wcstombs()*
204      atoi()       free()          mbstowcs()*    strtod()*       wctomb()*
205      atol()        getenv()       mbtowc()*

206  <string.h>
207  The header defines the macro

208      NULL

209  and the type

210      size_t

```

11 and declares the functions

12	<i>memchr()</i> *	<i>strcat()</i>	<i>strcspn()</i>	<i>strncmp()</i>	<i>strspn()</i>
13	<i>memcmp()</i> *	<i>strchr()</i>	<i>strerror()</i> *	<i>strncpy()</i>	<i>strstr()</i>
14	<i>memcpy()</i> *	<i>strcmp()</i>	<i>strlen()</i>	<i>strpbrk()</i>	<i>strtok()</i>
15	<i>memmove()</i> *	<i>strcoll()</i> *	<i>strncat()</i>	<i>strrchr()</i>	<i>strxfrm()</i> *
16	<i>memset()</i> *	<i>strcpy()</i>			

17 **<sys/stat.h>**

18 The header defines the macros

19	<i>S_IRGRP</i>	<i>S_IRWXO</i>	<i>S_ISCHR</i>	<i>S_ISGID</i>	<i>S_IWGRP</i>	<i>S_IXGRP</i>
20	<i>S_IROTH</i>	<i>S_IRWXU</i>	<i>S_ISDIR</i>	<i>S_ISREG</i>	<i>S_IWOTH</i>	<i>S_IXOTH</i>
21	<i>S_IRUSR</i>	<i>S_ISBLK</i>	<i>S_ISFIFO</i>	<i>S_ISUID</i>	<i>S_IWUSR</i>	<i>S_IXUSR</i>
22	<i>S_IRWXG</i>					

23 and declares the structure

24 *stat*

25 with structure elements

26	<i>st_atime</i>	<i>st_dev</i>	<i>st_ino</i>	<i>st_mtime</i>	<i>st_size</i>
27	<i>st_ctime</i>	<i>st_gid</i>	<i>st_mode</i>	<i>st_nlink</i>	<i>st_uid</i>

28 and the functions

29	<i>chmod()</i>	<i>mkdir()</i>	<i>stat()</i>
30	<i>fstat()</i>	<i>mkfifo()</i>	<i>umask()</i>

31 **<sys/times.h>**

32 The header defines the type

33 *clock_t*

34 and declares the structure

35 *tms*

36 with structure elements

37	<i>tms_cstime</i>	<i>tms_cutime</i>	<i>tms_stime</i>	<i>tms_utime</i>
----	-------------------	-------------------	------------------	------------------

38 and the function

39 *times()*

```

240      <sys/types.h>
241      The header defines the types
242          dev_t      ino_t      nlink_t      pid_t      ssize_t
243          gid_t      mode_t      off_t      size_t      uid_t
244      <sys/utsname.h>
245      The header declares the structure
246          utsname
247      with structure elements
248          machine      nodename      release      sysname      version
249      and the function
250          uname()
251      <sys/wait.h>
252      The header defines the macros
253          WEXITSTATUS      WIFSIGNALED      WNOHANG      WTERMSIG
254          WIFEXITED      WIFSTOPPED      WSTOPSIG      WUNTRACED
255      and declares the functions
256          wait()      waitpid()
257      <termios.h>
258      The header defines the macros
259          B0      B75      ECHONL      NCCS      TCSAFLUSH
260          B110      B9600      HUPCL      NOFLSH      TCSANOW
261          B1200      BRKINT      ICANON      OPOST      TOSTOP
262          B134      CLOCAL      ICRNL      PARENB      VEOF
263          B150      CREAD      IEXTEN      PARMRK      VEOL
264          B1800      CS5      IGNBRK      PARODD      VERASE
265          B19200      CS6      IGNCR      TCIFLUSH      VINTR
266          B200      CS7      IGNPAR      TCIOFF      VKILL
267          B2400      CS8      INLCR      TCIOFLUSH      VMIN
268          B300      CSIZE      INPCK      TCION      VQUIT
269          B38400      CSTOPB      ISIG      TCOFLUSH      VSTART
270          B4800      ECHO      ISTRIP      TCOOFF      VSTOP
271          B50      ECHOE      IXOFF      TCOON      VSUSP
272          B600      ECHOK      IXON      TCSADRAIN      VTIME

```

```

73  and the types
74      cc_t      speed_t  tcflag_t
75  and declares the structure
76      termios
77  with structure elements
78      c_cc      c_cflag  c_iflag  c_lflag  c_oflag
79  and the functions
80      cfgetispeed()  cfsetospeed()  tcflush()      tcsendbreak()
81      cfgetospeed()  tcdrain()      tcgetattr()    tcsetattr()
82      cfsetispeed()  tcflow()
83  <time.h>
84  The header defines the macros
85      CLK_TCK      CLOCKS_PER_SEC  NULL
86  the types
87      clock_t  size_t  time_t
88  and declares the structure
89      tm
90  with structure elements
91      tm_hour  tm_mday  tm_mon  tm_wday  tm_year
92      tm_isdst  tm_min  tm_sec  tm_yday
93  and the functions
94      asctime()  ctime()  gmtime()  mktime()  time()
95      clock()*  difftime()*  localtime()  strftime()  tzset()
96  and declares the external variable
97      tzname

```

298 **<unistd.h>**

299 The header defines the macros

300	F_OK	_PC_PIPE_BUF
301	NULL	_PC_VDISABLE
302	R_OK	_POSIX_CHOWN_RESTRICTED
303	SEEK_CUR	_POSIX_JOB_CONTROL
304	SEEK_END	_POSIX_NO_TRUNC
305	SEEK_SET	_POSIX_SAVED_IDS
306	STDERR_FILENO	_POSIX_VDISABLE
307	STDIN_FILENO	_POSIX_VERSION
308	STDOUT_FILENO	_SC_ARG_MAX
309	W_OK	_SC_CHILD_MAX
310	X_OK	_SC_CLK_TCK
311	_PC_CHOWN_RESTRICTED	_SC_JOB_CONTROL
312	_PC_LINK_MAX	_SC_NGROUPS_MAX
313	_PC_MAX_CANON	_SC_OPEN_MAX
314	_PC_MAX_INPUT	_SC_SAVED_IDS
315	_PC_NAME_MAX	_SC_STREAM_MAX
316	_PC_NO_TRUNC	_SC_TZNAME_MAX
317	_PC_PATH_MAX	_SC_VERSION

318 and defines the types

319 *size_t** *ssize_t**

320 and declares the functions

321	<i>_exit()</i>	<i>execl()</i>	<i>geteuid()</i>	<i>link()</i>	<i>setsid()</i>
322	<i>access()</i>	<i>execle()</i>	<i>getgid()</i>	<i>lseek()</i>	<i>setuid()</i>
323	<i>alarm()</i>	<i>execlp()</i>	<i>getgroups()</i>	<i>pathconf()</i>	<i>sleep()</i>
324	<i>chdir()</i>	<i>execv()</i>	<i>getlogin()</i>	<i>pause()</i>	<i>sysconf()</i>
325	<i>chown()</i>	<i>execve()</i>	<i>getpgrp()</i>	<i>pipe()</i>	<i>tcgetpgrp()</i>
326	<i>close()</i>	<i>execvp()</i>	<i>getpid()</i>	<i>read()</i>	<i>tcsetpgrp()</i>
327	<i>ctermid()</i>	<i>fork()</i>	<i>getppid()</i>	<i>rmdir()</i>	<i>ttyname()</i>
328	<i>cuserid()</i>	<i>fpathconf()</i>	<i>getuid()</i>	<i>setgid()</i>	<i>unlink()</i>
329	<i>dup2()</i>	<i>getcwd()</i>	<i>isatty()</i>	<i>setpgid()</i>	<i>write()</i>
330	<i>dup()</i>	<i>getegid()</i>			

331 NOTE: The *cuserid()* symbol is permitted in this header, but need not be supplied. See 2.7.2.

332 **<utime.h>**

333 The header declares the structure

334 *utimbuf*

335 with structure elements

<i>actime</i>	<i>modtime</i>	
and the function		
<i>utime()</i>		

Annex D (informative)

Profiles

1 This standard contains a number of options and variables that reflect the range of
2 systems and environments that might be encountered. In general, it will be use-
3 ful for applications to take the full range of these possibilities into account and
4 either accommodate them or exclude them. However, there are significant com-
5 munities of interest that may have common needs that warrant focusing on a
6 specific suite of these options and parameters. This annex discusses the concept
7 of *profiles* (also known as *functional standards*) and how they address this
8 problem.

9 This annex reflects current thinking. It is clear that a concept such as this will
10 help significantly in clarifying the intended use of these standards. It is to be
11 expected that some of the details of this material will be changed before it is fully
12 stabilized.

13 As background: the OSI model has over 170 standards (and consequent combina-
14 tions thereof) that fit within it. Only a fraction of those are actually useful for any
15 given application environment. The concept of *profiles* was developed to address
16 this issue and appears also to apply to the area of application portability. The
17 ISO/IEC term for such profiles is ISP, or "International Standardized Profile."

18 D.1 Definitions

19 The following definitions are proposed for use in the area covered by this part of
20 ISO/IEC 9945.

21 **D.1.1 Applications Environment Profile (AEP) [profile]:** The specification of
22 a complete and coherent subset of an Open System Environment, together with
23 the options and parameters necessary to support a class of applications for inter-
24 operability or applications portability, including consistency of data access and
25 human interfaces. Where there are several AEPs for the same OSE, they are har-
26 monized.

27 AEPs are the basis for procurement and conformance testing and are the target
28 environment for software development.

29 **D.1.2 Application Specific Environment (ASE):** A complete and coherent
30 subset of an Applications Environment Profile, together with interfaces, services,
31 or supporting formats outside of the profile, that are required by a particular

application for its installation and execution.

D.1.3 Application Specific Environment Description (ASED): The specification of an Application Specific Environment, together with the specific options or parameters required; interfaces, services, or supporting formats outside of the profile; and resource requirements necessary for the satisfactory operation of the application. (For example, storage and performance requirements.)

(This term is intended for use in Applications Conformance clauses found in profiles.)

D.1.4 coherent: The parts are logically connected. (For example, if both FORTRAN and COBOL are specified, whether they can share files is specified.)

D.1.5 complete: Having all the necessary parts. (For example, if COBOL and SQL are both specified, then there is a COBOL binding to SQL, or at least an explanation of why not.)

D.1.6 comprehensive: A sufficiently broad range of functionality is covered that the needs of most Applications Environment Profiles are met.

D.1.7 consistent: The parts of the Open System Environment do not inherently conflict with each other. This does not preclude options that conflict, as long as an Applications Environment Profile can select a set that does not conflict.

D.1.8 harmonized: Where same functionality is needed in several profiles, it appears identically in all of them.

D.1.9 Open System Environment (OSE): A comprehensive and consistent set of international information technology standards and functional standards (profiles) that specify interfaces, services, and supporting formats to accomplish interoperability and portability of applications, data, and people. These are based on International Standards (ISO, IEC, CCITT, ...)

D.1.10 POSIX Open System Environment: A comprehensive and consistent set of ISO/IEC, regional, and national information technology standards and functional standards (profiles) that specify interfaces, services, and supporting formats for interoperability and portability of applications, data, and people that are in accord with ISO/IEC 9945 (POSIX).

No single component of the OSE, including ISO/IEC 9945, is expected to be required in all such profiles.

D.2 Options in This Part of ISO/IEC 9945

In terms of this part of ISO/IEC 9945, there are a number of features that could be specified in a profile. This list includes:

- The options listed in 1.3.1.3.
- The limits in 2.8. Regarding the the C Language Limits for the type *char*, care should be taken that those limits are not for the POSIX.1 definition of *character*, but for the one in the C language. For the POSIX.1 definition of

- 71 *character*, the following limits from the C Standard {2} could be specified as
72 well: {MB_LEN_MAX} and {MB_CUR_MAX}.
- 73 — The flags in 2.9.4.
 - 74 — Instances of the word “may” throughout the document. (Note that not all
75 instances of “may” constitute behavior that could or should be considered
76 appropriate for specification in a profile. Some reflect implementation vari-
77 ants that should not matter to applications.)
 - 78 — Features that are specified in a generic way for broad portability of the
79 standard, that might reasonably be constrained in the more limited context
80 of a profile. For such features, the constraint shall be documented in a
81 profile so that the effects of the constraint on the standard would be
82 clarified.

83 D.3 Related Standards

84 The other POSIX standards (ISO/IEC 9945-2 {B36}, in particular) are expected to
85 form a major part of the POSIX OSE. Other formal standards, such as those listed
86 in Annex A, are also expected to be part of such a document (in particular, the
87 C Standard {2}).

88 Standards such as other languages, SQL, graphics standards such as GKS, and
89 networking standards are also probable candidates for inclusion in the POSIX
90 OSE.

91 D.4 Related Activities

92 In many ways, the work of NIST (in terms of FIPS), OSF, UNIX International, and
93 X/Open often act like early (but sophisticated) profiles or metaprofiles, specifying
94 a range of standards from which true profiles could select. They collect together
95 many standards, specify options, and specify the relationship between the parts.
96 These activities go well beyond profiles, as they add specifications that are not for-
97 mal standards to the suite as well. Often these additional specifications point to
98 areas where formal standards are required.

99 D.5 Relationship to IEEE Draft Project 1003.0

100 The IEEE P1003.0 working group is writing a *Guide to Open Systems*. In many
101 ways, this is the specification of the POSIX Open Systems Environment. It could
102 be the specification of the collection of standards that might be used to specify
103 many Open Systems Environments, depending on how exactly that work
104 proceeds.

THIS PAGE WAS
BLANK IN THE ORIGINAL

Annex E (informative)

Sample National Profile

1 One class of “community of interest” for which profiles (as discussed in Annex D)
2 are useful is specific countries, where the general characteristics warrant specific
3 focus to serve the needs of users in those countries. Such needs lead to a number
4 of implications concerning the options available within this part of ISO/IEC 9945
5 and may warrant specification of complementary standards as well.

6 The following is an example of a country’s needs with respect to this part of
7 ISO/IEC 9945 and how those needs relate to other international standards as well
8 as national standards. The example provided is included here for informative
9 purposes and is not a formal standard in the country in question. It is provided
10 by the Danish Standards Association and is as accurate as possible with regards
11 to Danish needs.¹⁾ This example national profile is worded as if it were a national
12 standard.

13 A subclass of conforming implementations can be identified that meet the require-
14 ments of a specific profile. By documenting these either in national standards, in
15 a document similar to an ISO/IEC ISP (an International Standardized Profile), or
16 in an informative annex (such as this), these can be referenced in a consistent
17 manner.

18 1) Further information may be obtained from the Danish Standards Association, Attn: S142u22A11
19 POSIX WG, Box 77, DK-2900 Hellerup, Denmark; FAX: +45 31 62 30 77; Email: posix@itc.dk

20 The data is also available electronically by anonymous FTAM or FTP at the site dkuug.dk in the
21 directory *isp*, where some other example national profiles, locales, and *chmaps* may also be
22 found. They are also available by an archive server reached at archive@dkuug.dk; use
23 “Subject: help” for further information.

24 More complete examples of profiles are expected to be available in future revisions of this part of
25 ISO/IEC 9945 and in other POSIX standards.

E.1 (Example) Profile for Denmark

NOTE: This profile is chosen both for its instructive value by being a European profile and the generality in the provisions it makes, addressing most of the relevant points. It does claim to be correct for Denmark, and the style is what would be expected in a real ISP. A collection of real ISPs would be as useful, and work is underway collecting these.

This is the definition of the Danish Standards Association POSIX.1 profile. Information on the actual data for the locale and coded character set mapping definitions are under development as part of an informative annex in ISO/IEC 9945-2 (B36).²⁾

The subset of conforming implementations that provide the required characteristics below is referred to as conforming to the “Danish Standards Association (DS) Environment Profile” for this part of ISO/IEC 9945.

The profile specifies no options according to POSIX.1 section 2.9.3. For section 2.8.4 in the `<limits.h>` specification, the `{_POSIX_TZNAME_MAX}` value shall be 7.

E.1.1 Character Encoding

Any character encoding with the required repertoire of the POSIX profile plus the following repertoire shall be allowed.

A “character set description file,” as described in ISO/IEC 9945-2, (B36) shall use the symbolic character names of the `ISO_10646_charmap` file described in the ISO/IEC 9945-2 (B36) sample profile annex for the characters encoded in the character set.

For the Danish and Greenlandic languages, the following characters shall be present in addition to the repertoire required by the POSIX profile: `<ae>`, `<o/>`, `<aa>`, `<AE>`, `<O/>`, and `<AA>`. For the Faroese language, the following characters shall be present in addition to the required POSIX locale characters and Danish repertoire: `<a'>`, `<i'>`, `<o'>`, `<u'>`, `<y'>`, `<d->`, `<A'>`, `<I'>`, `<O'>`, `<U'>`, `<Y'>`, and `<D->`.

Recommended character sets are ISO 8859-1 (B34) or ISO 10646 (B37). The **CHARSET** environment variable shall be used to specify the processing character set; for instance, `ISO_8859-1` or `ISO_10646`. This shall be used to select the encoded character-set-specific versions of the locale definitions. If no **CHARSET** variable is present, `ISO_8859-1` shall be assumed.

2) The 9945-2 document, “POSIX.2,” is currently in the state of a Committee Document (CD), to be approved as a Draft International Standard.

E.1.2 Character Encoding and Display

For terminal equipment not capable of generating or showing the processing character set, the character names defined in the current *charmap* file shall be used: characters in the *charmap* file having two-character names shall be specified by the two-character name preceded by the <intro> character, and characters having *charmap* names longer than two characters shall be specified by the character name preceded by the <intro> character and an <underline> and followed by an <underline>. In names longer than two characters, an <intro> character and an <underline> character in sequence shall signify a literal <underline> character part of the character name. Two <intro> characters in sequence shall signify one <intro> character, both in names and in the general stream.

For input, if the character name is undefined in the current *charmap* file, the data shall be left untouched (including the <intro> character) and the behavior is implementation defined.

E.1.3 Locale Definitions

The following guideline is used for specifying the locale identification string:³⁾

```
"%2.2s_%2.2s.%s,%s", <language>, <territory>, <character-set>,  
<version>
```

where <language> shall be taken from ISO 639 {B32} and <territory> shall be the two-letter country code of ISO 3166 {B33}, if possible. The <language> shall be specified with lowercase letters only, and the <territory> shall be specified in uppercase letters only. An optional <character-set> specification may follow after a <period> for the name of the character set; if just a numeric specification is present, this shall represent the number of the international standard describing the character set. If the <character-set> specification is not present, the encoded character set specific locale shall be determined by the **CHARSET** environment variable, and if this is unset or null, the encoding of ISO 8859-1 {B34} shall be assumed. A parameter specifying a <version> of the profile may be placed after the optional <character-set> specification, delimited by <comma>. This may be used to discriminate between different cultural needs; for instance, dictionary order versus a more systems-oriented collating order.

Following the above guidelines for locale names, the national Danish locale string shall be

```
da_DK
```

In the following, the TZ variable shall be specified according to the current official daylight-saving-time rules in Denmark. Since Daylight Saving Time is politically decided and thus changeable, this is only a recommendation.

3) The guideline was inspired by the *X/Open Portability Guide* {B61}. It is presented in the file format notation used by ISO/IEC 9945-2 {B36}.

The locale definition for Denmark shall be as follows:

LANG da_DK
TZ CET-1CET DST,M3.5.0/M9.5.0

The locale definition for the Faroe Islands shall be as follows:

LANG fo_DK
TZ UTC0UTC DST,M3.5.0/M9.5.0

The locale definition for Western Greenland shall be as follows:

LANG kl_DK
TZ UTZ+3VTZ,M3.5.0/M9.5.0

The locale definition for Eastern Greenland shall be as follows:

LANG kl_DK
TZ VTZ+2WTZ,M3.5.0/M9.5.0

For the Faroe Islands and Greenland, only the LC_TIME and LC_MESSAGES data are different from the Danish language specifications.

Identifier Index

<i>abort()</i>	Referenced C Language Routines {8.1}	151
<i>abs()</i>	Referenced C Language Routines {8.1}	151
<i>access()</i>	Check File Accessibility {5.6.3}	104
<i>acos()</i>	Referenced C Language Routines {8.1}	151
<i>alarm()</i>	Schedule Alarm {3.4.1}	63
<i>asctime()</i>	Extensions to Time Functions {8.1.1}	152
<i>asin()</i>	Referenced C Language Routines {8.1}	151
<i>assert()</i>	Referenced C Language Routines {8.1}	151
<i>atan()</i>	Referenced C Language Routines {8.1}	151
<i>atan2()</i>	Referenced C Language Routines {8.1}	151
<i>atof()</i>	Referenced C Language Routines {8.1}	151
<i>atoi()</i>	Referenced C Language Routines {8.1}	151
<i>atol()</i>	Referenced C Language Routines {8.1}	151
<i>bsearch()</i>	Referenced C Language Routines {8.1}	151
<i>calloc()</i>	Referenced C Language Routines {8.1}	151
<i>ceil()</i>	Referenced C Language Routines {8.1}	151
<i>cfgetispeed()</i>	Baud Rate Functions {7.1.3}	141
<i>cfgetospeed()</i>	Baud Rate Functions {7.1.3}	141
<i>cfsetispeed()</i>	Baud Rate Functions {7.1.3}	141
<i>cfsetospeed()</i>	Baud Rate Functions {7.1.3}	141
<i>chdir()</i>	Change Current Working Directory {5.2.1}	86
<i>chmod()</i>	Change File Modes {5.6.4}	106
<i>chown()</i>	Change Owner and Group of a File {5.6.5}	107
<i>clearerr()</i>	Referenced C Language Routines {8.1}	151
<i>close()</i>	Close a File {6.3.1}	115
<i>closedir()</i>	Directory Operations {5.1.2}	83
<i>cos()</i>	Referenced C Language Routines {8.1}	151
<i>cosh()</i>	Referenced C Language Routines {8.1}	151
<i>cpio</i>	Extended <i>cpio</i> Format {10.1.2}	173
<i>creat()</i>	Create a New File or Rewrite an Existing One {5.3.2}	91
<i>ctermid()</i>	Generate Terminal Pathname {4.7.1}	78
<i>ctime()</i>	Referenced C Language Routines {8.1}	152
<i>dev_t</i>	Primitive System Data Types {2.5}	27
<i>directory</i>	Directory Operations {5.1.2}	83
<i><dirent.h></i>	Format of Directory Entries {5.1.1}	83
<i>dup()</i>	Duplicate an Open File Descriptor {6.2.1}	114
<i>dup2()</i>	Duplicate an Open File Descriptor {6.2.1}	114
<i>environ</i>	Execute a File {3.1.2}	42
<i>errno</i>	Error Numbers {2.4}	23
<i><errno.h></i>	Error Numbers {2.4}	23
<i>exec</i>	Execute a File {3.1.2}	42
<i>execl()</i>	Execute a File {3.1.2}	42
<i>execle()</i>	Execute a File {3.1.2}	42
<i>execlp()</i>	Execute a File {3.1.2}	42
<i>execv()</i>	Execute a File {3.1.2}	42
<i>execve()</i>	Execute a File {3.1.2}	42

<i>execup()</i>	Execute a File {3.1.2}	42
<i>exit()</i>	Referenced C Language Routines {8.1}	151
<i>_exit()</i>	Terminate a Process {3.2.2}	49
<i>exp()</i>	Referenced C Language Routines {8.1}	151
<i>fabs()</i>	Referenced C Language Routines {8.1}	151
<i>fclose()</i>	Referenced C Language Routines {8.1}	151
<i>fcntl()</i>	File Control {6.5.2}	121
<i><fcntl.h></i>	Data Definitions for File Control Operations {6.5.1}	121
<i>fdopen()</i>	Open a Stream on a File Descriptor {8.2.2}	157
<i>feof()</i>	Referenced C Language Routines {8.1}	151
<i>ferror()</i>	Referenced C Language Routines {8.1}	151
<i>fflush()</i>	Referenced C Language Routines {8.1}	151
<i>fgetc()</i>	Referenced C Language Routines {8.1}	151
<i>fgets()</i>	Referenced C Language Routines {8.1}	151
<i>fileno()</i>	Map a Stream Pointer to a File Descriptor {8.2.1}	156
<i>floor()</i>	Referenced C Language Routines {8.1}	151
<i>fmod()</i>	Referenced C Language Routines {8.1}	151
<i>fopen()</i>	Referenced C Language Routines {8.1}	151
<i>fork()</i>	Process Creation {3.1.1}	41
<i>fpathconf()</i>	Get Configurable Pathname Variables {5.7.1}	110
<i>fprintf()</i>	Referenced C Language Routines {8.1}	151
<i>fputc()</i>	Referenced C Language Routines {8.1}	151
<i>fputs()</i>	Referenced C Language Routines {8.1}	151
<i>fread()</i>	Referenced C Language Routines {8.1}	151
<i>free()</i>	Referenced C Language Routines {8.1}	151
<i>freopen()</i>	Referenced C Language Routines {8.1}	151
<i>frexp()</i>	Referenced C Language Routines {8.1}	151
<i>fscanf()</i>	Referenced C Language Routines {8.1}	151
<i>fseek()</i>	Referenced C Language Routines {8.1}	151
<i>fstat()</i>	Get File Status {5.6.2}	103
<i>ftell()</i>	Referenced C Language Routines {8.1}	151
<i>fwrite()</i>	Referenced C Language Routines {8.1}	151
<i>getc()</i>	Referenced C Language Routines {8.1}	151
<i>getchar()</i>	Referenced C Language Routines {8.1}	151
<i>getcwd()</i>	Get Working Directory Pathname {5.2.2}	87
<i>getegid()</i>	Get Real User, Effective User, Real Group, and Effective Group IDs {4.2.1}	68
<i>getenv()</i>	Environment Access {4.6.1}	77
<i>geteuid()</i>	Get Real User, Effective User, Real Group, and Effective Group IDs {4.2.1}	68
<i>getgid()</i>	Get Real User, Effective User, Real Group, and Effective Group IDs {4.2.1}	68
<i>getgrgid()</i>	Group Database Access {9.2.1}	166
<i>getgrnam()</i>	Group Database Access {9.2.1}	166
<i>getgroups()</i>	Get Supplementary Group IDs {4.2.3}	70
<i>getlogin()</i>	Get User Name {4.2.4}	71
<i>getpgrp()</i>	Get Process Group ID {4.3.1}	72
<i>getpid()</i>	Get Process and Parent Process IDs {4.1.1}	67
<i>getppid()</i>	Get Process and Parent Process IDs {4.1.1}	67
<i>getpwnam()</i>	User Database Access {9.2.2}	167
<i>getpwuid()</i>	User Database Access {9.2.2}	167

<i>gets()</i>	Referenced C Language Routines (8.1)	151
<i>getuid()</i>	Get Real User, Effective User, Real Group, and Effective Group IDs (4.2.1)	68
<i>gid_t</i>	Primitive System Data Types (2.5)	27
<i>gmtime()</i>	Referenced C Language Routines (8.1)	152
<grp.h>	Group Database Access (9.2.1)	166
<i>ino_t</i>	Primitive System Data Types (2.5)	27
<i>isalnum()</i>	Referenced C Language Routines (8.1)	151
<i>isalpha()</i>	Referenced C Language Routines (8.1)	151
<i>isatty()</i>	Determine Terminal Device Name (4.7.2)	79
<i>iscntrl()</i>	Referenced C Language Routines (8.1)	151
<i>isdigit()</i>	Referenced C Language Routines (8.1)	151
<i>isgraph()</i>	Referenced C Language Routines (8.1)	151
<i>islower()</i>	Referenced C Language Routines (8.1)	151
<i>isprint()</i>	Referenced C Language Routines (8.1)	151
<i>ispunct()</i>	Referenced C Language Routines (8.1)	151
<i>isspace()</i>	Referenced C Language Routines (8.1)	151
<i>isupper()</i>	Referenced C Language Routines (8.1)	151
<i>isxdigit()</i>	Referenced C Language Routines (8.1)	151
<i>kill()</i>	Send a Signal to a Process (3.3.2)	56
<i>ldexp()</i>	Referenced C Language Routines (8.1)	151
<limits.h>	Numerical Limits (2.8)	34
<i>link()</i>	Link to a File (5.3.4)	92
<i>localtime()</i>	Referenced C Language Routines (8.1)	152
<i>log()</i>	Referenced C Language Routines (8.1)	151
<i>log10()</i>	Referenced C Language Routines (8.1)	151
<i>longjmp()</i>	Referenced C Language Routines (8.1)	151
<i>lseek()</i>	Reposition Read/Write File Offset (6.5.3)	127
<i>main()</i>	Description (3.1.2.2)	43
<i>malloc()</i>	Referenced C Language Routines (8.1)	151
<i>mkdir()</i>	Make a Directory (5.4.1)	94
<i>mkfifo()</i>	Make a FIFO Special File (5.4.2)	95
<i>mktime()</i>	Referenced C Language Routines (8.1)	152
<i>mode_t</i>	Primitive System Data Types (2.5)	27
<i>modf()</i>	Referenced C Language Routines (8.1)	151
<i>nlink_t</i>	Primitive System Data Types (2.5)	27
<i>off_t</i>	Primitive System Data Types (2.5)	27
<i>open()</i>	Open a File (5.3.1)	88
<i>opendir()</i>	Directory Operations (5.1.2)	83
<i>pathconf()</i>	Get Configurable Pathname Variables (5.7.1)	110
<i>pause()</i>	Suspend Process Execution (3.4.2)	64
<i>perror()</i>	Referenced C Language Routines (8.1)	151
<i>pid_t</i>	Primitive System Data Types (2.5)	27
<i>pipe()</i>	Create an Inter-Process Channel (6.1.1)	113
<i>pow()</i>	Referenced C Language Routines (8.1)	151
<i>printf()</i>	Referenced C Language Routines (8.1)	151
<i>putc()</i>	Referenced C Language Routines (8.1)	151
<i>putchar()</i>	Referenced C Language Routines (8.1)	151
<i>puts()</i>	Referenced C Language Routines (8.1)	151
<pwd.h>	User Database Access (9.2.2)	167
<i>qsort()</i>	Referenced C Language Routines (8.1)	151

<i>rand()</i>	Referenced C Language Routines {8.1}	151
<i>read()</i>	Read from a File {6.4.1}	116
<i>readdir()</i>	Directory Operations {5.1.2}	83
<i>realloc()</i>	Referenced C Language Routines {8.1}	151
<i>remove()</i>	Referenced C Language Routines {8.1}	151
<i>rename()</i>	Rename a File {5.5.3}	99
<i>rewind()</i>	Referenced C Language Routines {8.1}	151
<i>rewinddir()</i>	Directory Operations {5.1.2}	83
<i>rmdir()</i>	Remove a Directory {5.5.2}	98
<i>scanf()</i>	Referenced C Language Routines {8.1}	151
<i>setbuf()</i>	Referenced C Language Routines {8.1}	151
<i>setgid()</i>	Set User and Group IDs {4.2.2}	68
<i>setjmp()</i>	Referenced C Language Routines {8.1}	151
<i>setlocale()</i>	Extensions to <i>setlocale()</i> Function {8.1.2}	154
<i>setpgid()</i>	Set Process Group ID for Job Control {4.3.3}	73
<i>setsid()</i>	Create Session and Set Process Group ID {4.3.2}	72
<i>setuid()</i>	Set User and Group IDs {4.2.2}	68
<i>sigaction()</i>	Examine and Change Signal Action {3.3.4}	58
<i>sigaddset()</i>	Manipulate Signal Sets {3.3.3}	57
<i>sigdelset()</i>	Manipulate Signal Sets {3.3.3}	57
<i>sigemptyset()</i>	Manipulate Signal Sets {3.3.3}	57
<i>sigfillset()</i>	Manipulate Signal Sets {3.3.3}	57
<i>sigismember()</i>	Manipulate Signal Sets {3.3.3}	57
<i>siglongjmp()</i>	Nonlocal Jumps {8.3.1}	162
<signal.h>	Signal Concepts {3.3.1}	51
<i>sigpending()</i>	Examine Pending Signals {3.3.6}	62
<i>sigprocmask()</i>	Examine and Change Blocked Signals {3.3.5}	60
<i>sigsetjmp()</i>	Nonlocal Jumps {8.3.1}	162
<i>sigsetops</i>	Manipulate Signal Sets {3.3.3}	57
<i>sigsuspend()</i>	Wait for a Signal {3.3.7}	62
<i>sin()</i>	Referenced C Language Routines {8.1}	151
<i>sinh()</i>	Referenced C Language Routines {8.1}	151
<i>size_t</i>	Primitive System Data Types {2.5}	27
<i>sleep()</i>	Delay Process Execution {3.4.3}	65
<i>sprintf()</i>	Referenced C Language Routines {8.1}	151
<i>sqrt()</i>	Referenced C Language Routines {8.1}	151
<i>srand()</i>	Referenced C Language Routines {8.1}	151
<i>sscanf()</i>	Referenced C Language Routines {8.1}	151
<i>ssize_t</i>	Primitive System Data Types {2.5}	27
<i>stat()</i>	Get File Status {5.6.2}	103
<i>strcat()</i>	Referenced C Language Routines {8.1}	151
<i>strchr()</i>	Referenced C Language Routines {8.1}	151
<i>strcmp()</i>	Referenced C Language Routines {8.1}	151
<i>strcpy()</i>	Referenced C Language Routines {8.1}	151
<i>strcspn()</i>	Referenced C Language Routines {8.1}	151
<i>strftime()</i>	Referenced C Language Routines {8.1}	152
<i>strlen()</i>	Referenced C Language Routines {8.1}	151
<i>strncat()</i>	Referenced C Language Routines {8.1}	151
<i>strncmp()</i>	Referenced C Language Routines {8.1}	151
<i>strncpy()</i>	Referenced C Language Routines {8.1}	151
<i>strpbrk()</i>	Referenced C Language Routines {8.1}	151

<i>strchr()</i>	Referenced C Language Routines {8.1}	151
<i>strspn()</i>	Referenced C Language Routines {8.1}	151
<i>strstr()</i>	Referenced C Language Routines {8.1}	151
<i>strtok()</i>	Referenced C Language Routines {8.1}	151
<i>sysconf()</i>	Get Configurable System Variables {4.8.1}	80
<sys/stat.h>	File Characteristics: Header and Data Structure {5.6.1} ...	101
<sys/times.h>	Get Process Times {4.5.2}	76
<sys/types.h>	Primitive System Data Types {2.5}	27
<sys/utsname.h>	Get System Name {4.4.1}	74
<sys/wait.h>	Wait for Process Termination {3.2.1}	47
<i>tan()</i>	Referenced C Language Routines {8.1}	151
<i>tanh()</i>	Referenced C Language Routines {8.1}	151
<i>tar</i>	Extended tar Format {10.1.1}	169
<i>tcdrain()</i>	Line Control Functions {7.2.2}	145
<i>tcflow()</i>	Line Control Functions {7.2.2}	145
<i>tcflush()</i>	Line Control Functions {7.2.2}	145
<i>tcgetattr()</i>	Get and Set State {7.2.1}	143
<i>tcgetpgrp()</i>	Get Foreground Process Group ID {7.2.3}	147
<i>tcsendbreak()</i>	Line Control Functions {7.2.2}	145
<i>tcsetattr()</i>	Get and Set State {7.2.1}	143
<i>tcsetpgrp()</i>	Set Foreground Process Group ID {7.2.4}	148
<i>termios</i>	General Terminal Interface {7.1}	129
<termios.h>	Parameters That Can Be Set {7.1.2}	135
<i>time()</i>	Get System Time {4.5.1}	75
<i>times()</i>	Get Process Times {4.5.2}	76
<i>time_t</i>	Primitive System Data Types {2.5}	27
<i>tmpfile()</i>	Referenced C Language Routines {8.1}	151
<i>tmpnam()</i>	Referenced C Language Routines {8.1}	151
<i>tolower()</i>	Referenced C Language Routines {8.1}	151
<i>toupper()</i>	Referenced C Language Routines {8.1}	151
<i>ttyname()</i>	Determine Terminal Device Name {4.7.2}	79
<i>tzset()</i>	Set Time Zone {8.3.2}	162
<i>uid_t</i>	Primitive System Data Types {2.5}	27
<i>umask()</i>	Set File Creation Mask {5.3.3}	91
<i>uname()</i>	Get System Name {4.4.1}	74
<i>ungetc()</i>	Referenced C Language Routines {8.1}	151
<unistd.h>	Symbolic Constants {2.9}	37
<i>unlink()</i>	Remove Directory Entries {5.5.1}	96
<i>utime()</i>	Set File Access and Modification Times {5.6.6}	108
<i>wait</i>	Wait for Process Termination {3.2.1}	47
<i>waitpid()</i>	Wait for Process Termination {3.2.1}	47
<i>write()</i>	Write to a File {6.4.2}	118

THIS PAGE WAS
BLANK IN THE ORIGINAL

Alphabetic Topical Index

A

Abbreviations ... 20, 208
abort() ... 46, 158, 160-161, 291, 307
 definition of ... 151
 rationale ... 291
abs() ... 307
 definition of ... 151
 absolute pathname
 definition of ... 11
 access control lists ... 209
 access mode
 definition of ... 11
access() ... 37, 104-105, 208, 261, 311
 definition of ... 104
 rationale ... 261
 symbolic constants ... 37
acos() ... 305
 definition of ... 151
actime ... 109, 312
 address space
 definition of ... 11
 AEP (see Applications Environment Profile)
 alarm
 schedule ... 63
alarm() ... 42, 46, 63-66, 241, 244-246, 272, 311
 definition of ... 63
 rationale ... 244
 ANSI ... 300
 Application Conformance ... 4, 192
 Application Specific Environment (ASE)
 definition of ... 313
 Application Specific Environment Description (ASED)
 definition of ... 314
 Applications Environment Profile (AEP)
 [profile]
 definition of ... 313
 appropriate privileges ... 17, 21, 26, 39, 56, 69-70, 92-93, 97, 105-111, 169, 172, 176-177, 199, 208, 246-247, 261
 definition of ... 11
 rationale ... 199
 Archive/Interchange File Format ... 169, 294
 {ARG_MAX} ... 24, 29, 44-45, 80, 230, 304
 definition of ... 36

array of *char*
 definition of ... 29
 ASCII ... 207
asctime() ... 310
 definition of ... 152
 ASED (see Application Specific Environment Description)
asin() ... 305
 definition of ... 151
_asm_built_in_atoi() ... 195
assert() ... 301
 definition of ... 151
 <assert.h> ... 301
 asynchronous I/O ... 289
atan2() ... 305
 definition of ... 151
atan() ... 305
 definition of ... 151
atexit() ... 189, 232, 307
atof() ... 307
 definition of ... 151
atoi() ... 194-195, 307
 definition of ... 151
atol() ... 307
 definition of ... 151

B

background process group
 definition of ... 12
 background process
 definition of ... 12
 background ... 12, 15, 51, 118, 121, 130-131, 139, 143, 147, 202-204, 206, 248, 277-278, 281-282
 Base by POSIX.1, Additions by the C Standard ... 188
 Base by the C Standard
 Additions by POSIX.1 ... 188
 Baud Rate Functions ... 141, 281
 Baud Rate Values ... 141, 281
 Bibliography ... 179
 binary stream ... 152
 block special file
 definition of ... 12

- blocked signals ... 60
- brk()* ... 188, 292
 - rationale for omission ... 292
- BRKINT ... 136
- BSD ... 199, 202, 205-207, 209, 213-214, 216, 222, 231-234, 236-239, 241-244, 247-250, 253, 258-260, 262-264, 266, 270, 272-273, 276-278, 282-284, 287, 291, 294, 296
 - signals ... 234
- bsearch()* ... 188, 307
 - definition of ... 151
- By Neither POSIX.1 Nor the C Standard ... 188
- byte
 - definition of ... 29
- Byte-Oriented *cpio* Archive Entry ... 174
- C**
- C Language Definitions ... 29, 217
- C Language Input/Output Functions ... 155, 287
- C Language Limits ... 34, 223
- C language
 - FILE functions ... 158
 - input/output functions ... 155
 - language-dependent services ... 5
 - related specifications ... 6
- C locale ... 154-155, 286
- C Shell ... 202-204, 216
- C Standard Language-Dependent Support ... 32
- C Standard Language-Dependent System Support ... 5, 194
- C Standard ... 2, 5-7, 12, 20, 24, 26-27, 29-30, 32-35, 46, 51, 54-55, 60-61, 103, 134, 151-152, 154-157, 159, 161-162, 186-190, 192-195, 197, 199-200, 212-214, 216-218, 220-223, 228-229, 232-235, 239, 243, 245-246, 250, 259, 266, 270, 272, 275, 285-289, 291, 301, 305, 315
 - abbreviation ... 20
 - definition of ... 20
 - symbols ... 29
- calloc()* ... 307
 - definition of ... 151
- Canonical Mode Input Processing ... 132, 278
- case folding ... 209-210
- c_cc* ... 132, 140-141, 310
- c_flag* ... 135, 138, 310
- cc_t* ... 310
 - definition of ... 135
- ceil()* ... 305
 - definition of ... 151
- cfgetispeed()* ... 33, 141-142, 310
 - definition of ... 141
 - rationale ... 281
- cfgetospeed()* ... 33, 141-142, 310
 - definition of ... 141
 - rationale ... 281
- cfsetispeed()* ... 33, 141-142, 144, 281, 310
 - definition of ... 141
 - rationale ... 281
- cfsetospeed()* ... 33, 141-142, 310
 - definition of ... 141
 - rationale ... 281
- Change Current Working Directory ... 86, 256
- Change File Modes ... 106, 261
- Change Owner and Group of a File ... 107
- Change Owner and Group of File ... 262
- Change Process's Root Directory ... 257
- Character Encoding and Display ... 319
- Character Encoding ... 318
- character set
 - portable filename ... 207
- character special file
 - definition of ... 12
- character
 - definition of ... 12
- {CHAR_BIT} ... 34, 304
- {CHAR_MAX} ... 34, 304
- {CHAR_MIN} ... 34, 223, 304
- CHARSET**
 - variable ... 318-319
- chdir()* ... 86-87, 176, 256, 261, 311
 - definition of ... 86
 - rationale ... 256
- Check File Accessibility ... 104
- child process
 - definition of ... 12
- {CHILD_MAX} ... 80, 215, 224, 304
 - definition of ... 36
- chmod()* ... 21, 33, 46, 92, 95-96, 103, 105-106, 108, 177, 261, 308
 - definition of ... 106
 - rationale ... 261
- chown()* ... 39, 103, 107-108, 253, 262-263, 311
 - definition of ... 107
 - rationale ... 262
- chroot()* ... 207, 257
 - rationale for omission ... 257

- c_iflag* ... 131, 136, 310
- clearenv()* ... 250
- clearerr()* ... 307
 - definition of ... 151
- c_lflag* ... 131, 139, 310
- [CLK_TCK] ... 81, 252
 - definition of ... 81
- CLOCAL ... 129, 135, 138
- clock tick ... 12, 29, 76-77, 80, 199, 217, 244, 249-251
 - definition of ... 12
 - rationale ... 199
- clock()* ... 199, 310
- CLOCKS_PER_SEC ... 310
- clock_t* ... 12, 29, 76-77, 249-250, 308, 310
 - definition of ... 29
- Close a File ... 115, 266
- close()* ... 45, 50, 91, 97, 115-116, 127, 158, 160, 279, 300, 311
 - definition of ... 115
 - rationale ... 266
- closedir()* ... 33, 83-85, 254-255, 302
 - definition of ... 83
 - rationale ... 254
- Closing a Terminal Device File ... 135, 279
- cmd*
 - Values for *fcntl()* ... 122
- c_oflag* ... 133, 137, 310
- coherent
 - definition of ... 314
- command interpreter
 - portable ... 231
- Common-Usage C Language-Dependent System Support ... 6, 194
- Common-Usage-Dependent Support ... 32
- Compile-Time Symbolic Constants for Portability Specifications ... 38, 225
- Compile-Time Symbolic Constants ... 38
- complete
 - definition of ... 314
- compliance ... 189
- comprehensive
 - definition of ... 314
- Configurable Pathname Variables ... 110-111, 262
- Configurable System Variables ... 80, 251
- configurable variables ... 80
- conformance document
 - definition of ... 10
 - rationale ... 197
- conformance ... 2-7, 9-11, 23, 33, 59, 109, 129, 151, 187-190, 192, 194-197, 199, 211, 213, 218, 222, 283, 313-314
 - application ... 4
 - implementation ... 2
- conforming application ... 3-5, 11, 188, 190-192, 216, 235-236, 246, 264-265, 267
 - strictly ... 2-4, 10, 28, 34, 38, 144, 190, 192, 196, 228, 239, 244, 264
- Conforming Implementation Options ... 4, 191
- Conforming POSIX.1 Application Using Extensions ... 5, 192
- Conforming POSIX.1 Application ... 4, 192
- consistent
 - definition of ... 314
- constants
 - compile-time ... 38
 - execution-time ... 38
 - symbolic ... 37
- contiguous file ... 294, 297-298
- Control Modes ... 138, 280
- Control Operations on Files ... 121, 270
- controlling process
 - definition of ... 12
- Controlling Terminal ... 130, 276
- controlling terminal
 - definition of ... 12
 - rationale ... 199
- Conventions ... 9, 196
- cooked mode ... 273
- core ... 209, 230-232
- cos()* ... 305
 - definition of ... 151
- cosh()* ... 305
 - definition of ... 151
- covert channel ... 211, 242
- cpio* ... 173-174, 176, 294-299
 - Archive Entry ... 174
 - File Data ... 176, 298
 - File Name ... 175, 298
 - format ... 173
 - Header ... 174, 298
 - Special Entries ... 176, 298
 - Values ... 176, 298
- CREAD ... 138
- creat()* ... 21, 33, 91-92, 103-104, 115-116, 118, 121, 128, 155, 169, 193, 257-258, 287, 303
 - definition of ... 91
 - rationale ... 258
- Create a New File or Rewrite an Existing One ... 91, 258
- Create an Inter-Process Channel ... 113, 265
- Create Session and Set Process Group ID ... 72, 248

Cross-References ... 42, 46, 49-50, 57-58, 60,
62-68, 70-74, 77-79, 85-87, 91-93, 95-97, 99,
101, 103-105, 107-108, 110, 114-116, 118,
121, 127-128, 142, 145, 147-148, 156-157,
162, 166-167, 173, 177
CR ... 134, 137
CRT ... 134
CSIZE ... 138
CSTOPB ... 138
ctermid() ... 33, 78-79, 188, 251, 311
 definition of ... 78
 rationale ... 251
ctime() ... 152-153, 162, 310
 definition of ... 152
<ctype.h> ... 301
currency symbols ... 295
current working directory
 change ... 86
 definition of ... 12, 20
cuserid() ... 247, 311
 rationale ... 247

D

Data Definitions for File Control Operations
 ... 121, 270
Data Interchange Format ... 169, 294
Database Access ... 166, 293
Database Standards ... 181
database
 group ... 165
 user ... 165
Definitions and General Requirements
 ... 196
Definitions ... 10, 197, 313
Delay Process Execution ... 65, 245
Denmark example profile ... 318
Determine Terminal Device Name ... 79, 251
/dev ... 297
Device- and Class-Specific Functions ... 129,
 273
device number ... 199
device
 definition of ... 13
 logical ... 206
dev_t ... 101, 214, 309
 definition of ... 27
/dev/tty ... 206, 251
difftime() ... 310
d_ino ... 253

direct ... 253
Directories ... 83, 253
directory entry
 definition of ... 13
 format ... 83
 rationale ... 199
 remove ... 96
Directory Operations ... 83, 254
directory ... 199
 change current working ... 86
 current working ... 12
 definition of ... 13
 empty ... 13
 make ... 94
 parent ... 16
 remove ... 98
 root ... 19, 207
 working ... 20
 working pathname ... 87
dirent ... 83, 85, 253-254, 302
<*dirent.h*> ... 83-85, 253, 302
 definition of ... 83
DIR ... 83-85, 255, 302
 definition of ... 84
div() ... 307
div_t ... 307
d_name ... 83, 254, 302
document
 conformance ... 10
Documentation ... 3, 190
documentation
 system ... 11
dot ... 13-14, 23, 26, 84, 98, 100, 192, 199,
 216, 254, 260
 definition of ... 13
dot-dot ... 13-14, 16, 23, 26, 84, 98, 100, 192,
 199, 207, 211, 254, 260
 definition of ... 13
dup2() ... 114-115, 265, 311
 definition of ... 114
 rationale ... 265
dup() ... 91, 104, 114-116, 118, 121, 128, 158,
 265, 287, 311
 definition of ... 114
 rationale ... 265
Duplicate an Open File Descriptor ... 114,
 265

E

[E2BIG] ... 24, 45, 230, 302
 definition of ... 24

- [EACCES] ... 24, 45, 74, 85-87, 90, 93-94, 96-98, 100, 104-106, 108-109, 112, 249, 256-257, 302
definition of ... 24
- eaccess()* ... 261
- [EAGAIN] ... 24, 42, 117-118, 120, 131, 227, 264, 267-270, 302
definition of ... 24
- [EBADF] ... 24, 85, 104, 112, 114-116, 118, 120, 126, 128, 145-149, 265, 267, 270, 302
definition of ... 24
- [EBUSY] ... 24, 97-98, 100, 213, 259-260, 302
definition of ... 24
- [ECHILD] ... 24, 49, 302
definition of ... 24
- ECHOE ... 139
- ECHOK ... 139
- ECHONL ... 139
- ECHO ... 139
- ed* ... 260
- [EDEADLK] ... 24, 125, 127, 272, 302
definition of ... 24
- [EDOM] ... 24, 213, 302
definition of ... 24
- [EEXIST] ... 24, 90, 93-94, 96, 98, 100, 259, 302
definition of ... 24
- [EFAULT] ... 24, 212, 230, 302
definition of ... 24
- [EFBIG] ... 24, 121, 302
definition of ... 24
- effective group ID ... 14-15, 19-20, 39, 44, 68-69, 88, 94-95, 102, 105-107, 229, 246-247, 253, 262
definition of ... 13
- effective user ID ... 14, 19-20, 44, 56, 68-69, 88, 94-95, 102, 105-110, 208, 229, 241, 246, 261
definition of ... 13
- Eighth Edition UNIX ... 198, 266, 269
- [EINTR] ... 24, 49, 63, 65, 90, 115-119, 121, 124, 126, 145-146, 213, 227, 230, 234, 240-241, 266, 302
definition of ... 24
- [EINVAL] ... 25, 49, 57-58, 60-61, 69-70, 74, 81, 87, 100, 105, 108, 112, 126, 128, 144-145, 147, 149, 243, 263, 272, 302
definition of ... 25
- [EIO] ... 25, 118, 121, 130-131, 135, 302
definition of ... 25
- [EISDIR] ... 25, 90, 100, 302
definition of ... 25
- [EMFILE] ... 25, 85, 90, 114-115, 126, 302
definition of ... 25
- [EMLINK] ... 25, 93-94, 100, 302
definition of ... 25
- empty directory
definition of ... 13
- empty string
definition of ... 13
- [ENAMETOOLONG] ... 25, 45, 85-86, 90, 93, 95-98, 100, 104-106, 108-109, 112, 230, 258, 260, 302
definition of ... 25
- endgrent()* ... 293
rationale for omission ... 293
- endpwent()* ... 293
- [ENFILE] ... 25, 85, 90, 114, 302
definition of ... 25
- [ENODEV] ... 25, 302
definition of ... 25
- [ENOENT] ... 25, 46, 85-86, 90, 93, 95-97, 99-100, 104-105, 107-108, 110, 112, 302
definition of ... 25
- [ENOEXEC] ... 25, 46, 228, 230, 302
definition of ... 25
- [ENOLCK] ... 25, 126, 271, 302
definition of ... 25
- [ENOMEM] ... 25, 42, 46, 213, 227, 291, 302
definition of ... 25
- [ENOSPC] ... 26, 90, 93, 95-96, 100, 121, 302
definition of ... 26
- [ENOSYS] ... 26, 74, 148-149, 197, 302
definition of ... 26
- [ENOTDIR] ... 26, 46, 85-86, 90, 93, 95-97, 99, 101, 104-106, 108, 110, 112, 302
definition of ... 26
- [ENOTEMPTY] ... 26, 98, 100, 259-260, 302
definition of ... 26
- [ENOTTY] ... 26, 145-149, 213, 274-275, 302
definition of ... 26
- environ*
definition of ... 42
- Environment Access ... 77, 250
- Environment Description ... 27, 216
- [ENXIO] ... 26, 90, 302
definition of ... 26
- EOF ... 132, 134, 139
- EOL ... 132, 134, 139
- [EPERM] ... 26, 57, 69-70, 73-74, 93, 97, 107-108, 110, 149, 242, 302
definition of ... 26
- [EPIPE] ... 26, 121, 213, 302
definition of ... 26

Epoch ... 19, 76, 103, 109, 152, 200, 208, 284
definition of ... 13

[ERANGE] ... 26, 87, 213, 257, 302
definition of ... 26

ERASE ... 132, 134, 139

[EROFS] ... 26, 90, 93, 95-97, 99, 101, 105,
107-108, 110, 213, 302
definition of ... 26

errno ... 211, 302
definition of ... 23

<errno.h> ... 23, 217, 259, 302
definition of ... 23

[ERRNO] ... 9

Error Numbers ... 23, 211

Error Reporting ... 161, 290

[ESPIPE] ... 26, 128, 302
definition of ... 26

[ESRCH] ... 26, 57, 74, 241-242, 302
definition of ... 26

/etc/passwd ... 209

[ETXTBSY] ... 230

[EWOULDBLOCK] ... 264

Examine and Change Blocked Signals ... 60,
244

Examine and Change Signal Action ... 58,
243

Examine Pending Signals ... 62, 244

example profile ... 318

[EXDEV] ... 26, 93, 101, 260, 302
definition of ... 26

exec ... 34-35, 42-43, 45-46, 59, 64, 67, 70,
73-74, 77, 84, 91, 96, 102-103, 115-116, 123,
127, 158-159, 226, 228-231, 246, 248-249,
254, 261, 266, 271, 287, 289
definition of ... 42
of shell scripts ... 228
rationale ... 228

execl() ... 42-44, 46, 228, 311
definition of ... 42

execle() ... 42-44, 46, 228, 311
definition of ... 42

execlp() ... 42-44, 228, 311
definition of ... 42

Execute a File ... 42, 228

Execution-Time Symbolic Constants for Portability Specifications ... 38, 225

Execution-Time Symbolic Constants ... 39

execv() ... 42, 44, 46, 228, 311
definition of ... 42

execve() ... 42, 44, 46, 228, 311
definition of ... 42

execvp() ... 42, 44, 228, 311
definition of ... 42

exit() ... 46, 48, 152, 158, 160-161, 189, 232-
233, 240, 266, 288-289, 291, 307
definition of ... 151
rationale ... 291

EXIT_FAILURE ... 152

EXIT_SUCCESS ... 152, 233

_exit() ... 46, 48-50, 73, 158-159, 189, 232-
233, 240, 288-289, 311
definition of ... 49
rationale ... 232

exp() ... 305
definition of ... 151

Extended *cpio* Format ... 173, 297

extended security controls
definition of ... 21, 208

Extended *tar* Format ... 169, 296

Extensions to *setlocale()* Function ... 154,
285

Extensions to Time Functions ... 152, 283

F

fabs() ... 305
definition of ... 151

Fast File System ... 262

fclose() ... 158, 160-161, 189, 289, 291, 307
definition of ... 151
rationale ... 289

fcntl() ... 33, 42, 46, 91, 104, 113-116, 118,
121, 123-128, 131, 158, 176, 214, 265, 268,
270-271, 275, 287, 303
definition of ... 121
rationale ... 270
Return Values ... 126

<fcntl.h> ... 30, 88, 91, 121, 123-124, 127,
302
definition of ... 121

fdopen() ... 33, 156-158, 188, 288, 307
definition of ... 157
rationale ... 287

FD_CLOEXEC ... 44, 88, 113, 122-123, 158,
254, 270, 302

F_DUPFD ... 114, 122-123, 126, 270, 302

feature test macro
definition of ... 13

feature test macros
suggested ... 222

feof() ... 159, 307
definition of ... 151

- ferror()* ... 307
 - definition of ... 151
- fflush()* ... 159-161, 288-290, 307
 - definition of ... 151
 - rationale ... 290
- fgetc()* ... 160, 290, 307
 - definition of ... 151
 - rationale ... 290
- F_GETFD ... 122-123, 125, 270, 302
- F_GETFL ... 122-123, 125, 270, 302
- F_GETLK ... 122, 124-126, 302
- fgetpos()* ... 214, 272, 307
- fgets()* ... 160, 290, 307
 - definition of ... 151
 - rationale ... 290
- FIFO special file ... 201
 - definition of ... 13
- FIFO ... 13-14, 17, 26, 88-90, 95, 102, 111, 116-117, 119-121, 128, 170, 172-173, 175-176, 201, 207, 253, 257, 259, 267-269
 - make ... 95
- File Access Modes Used for *open()* and *fcntl()* ... 123
- file access permissions
 - definition of ... 21, 208
- File Accessibility ... 261
- File Characteristics: Header and Data Structure ... 101, 260
- File Characteristics ... 101, 260
- file classes ... 201
- File Control ... 121, 270
- file description
 - definition of ... 14
- File Descriptor Deassignment ... 115, 266
- File Descriptor Flags Used for *fcntl()* ... 122
- File Descriptor Manipulation ... 114, 265
- file descriptor
 - definition of ... 14
 - duplicate ... 114, 122
 - map a stream pointer ... 156
 - open stream ... 157
- file group class
 - definition of ... 14
- file hierarchy
 - definition of ... 22, 208
- file mode
 - definition of ... 14
- file offset
 - definition of ... 14
- file other class
 - definition of ... 14
- file owner class
 - definition of ... 14
- file permission bits ... 14, 21, 88, 91, 94-95, 103, 105-106, 261
 - definition of ... 15
- file permissions ... 13-16, 20-21, 24, 27, 88, 90-92, 94-95, 103, 105-106, 169, 171-172, 174-175, 208, 246, 261, 263, 277
 - definition of ... 209
- File Removal ... 96, 259
- file serial number ... 101
 - definition of ... 15
- File Status Flags Used for *open()* and *fcntl()* ... 122
- file system ... 201
 - definition of ... 15
 - mounted ... 206
 - read-only ... 18
 - root ... 208
 - root of ... 208
- file times update
 - definition of ... 22, 210
- file type (see *file*)
- file ... 201
 - access ... 104
 - access permissions ... 21
 - binary ... 152, 187
 - block special ... 12
 - change modes ... 106
 - change owner and group ... 107
 - character special ... 12
 - characteristics header ... 101
 - close ... 115
 - contiguous ... 294, 297-298
 - control ... 121
 - create or rewrite ... 91
 - definition of ... 14
 - execute ... 42
 - FIFO special ... 13
 - get status ... 103
 - hierarchy ... 22
 - high performance ... 294, 297-298
 - link to ... 92
 - locking ... 124, 270
 - make FIFO special ... 95
 - mode ... 14
 - offset ... 14
 - open ... 88
 - passwd ... 207
 - permission bits ... 15
 - read ... 116
 - regular ... 19
 - remove ... 162
 - rename ... 99
 - reposition read/write offset ... 127
 - serial number ... 15
 - set access and modification times ... 108

set creation mask ... 91
 text ... 152, 187
 times ... 108
 times update ... 22
 write ... 118
 filename portability
 definition of ... 22, 209
 filename ... 201
 definition of ... 14
fileno() ... 33, 156, 158, 188, 206, 307
 definition of ... 156
 rationale ... 287
 Files and Directories ... 83, 253
FILE ... 156-158, 160, 287-288, 307
<float.h> ... 303
flock ... 124, 302
 Structure ... 125
floor() ... 305
 definition of ... 151
fmod() ... 305
 definition of ... 151
F_OK ... 37-38, 105, 311
fopen() ... 157-158, 160-161, 201, 288-289,
 291, 307
 definition of ... 151
 rationale ... 289
 {FOPEN_MAX} ... 35
 foreground process group ID
 definition of ... 15
 foreground process group
 definition of ... 15
 foreground process
 definition of ... 15
 foreground ... 12, 15, 129-131, 133-134, 136,
 140, 143, 147-148, 202-206, 248, 275-278,
 282-283
fork() ... 17-18, 41-42, 46, 49, 64, 67, 73, 77,
 84, 116, 125, 130, 156, 158-159, 203, 214,
 226-227, 229-231, 248, 254, 270-271, 276,
 287-289, 311
 definition of ... 41
 rationale ... 226
 Format of Directory Entries ... 83, 253
fpathconf() ... 39, 110-112, 263, 311
 definition of ... 110
 rationale ... 263
fpos_t ... 306
fprintf() ... 161, 272, 307
 definition of ... 151
fputc() ... 160, 290, 307
 definition of ... 151
 rationale ... 290

fputs() ... 160, 290, 307
 definition of ... 151
 rationale ... 290
F_RDLCK ... 122, 124-126, 302
fread() ... 160, 290, 307
 definition of ... 151
 rationale ... 290
free() ... 188, 240, 256, 307
 definition of ... 151
freopen() ... 158, 160, 290, 307
 definition of ... 151
 rationale ... 290
frexp() ... 305
 definition of ... 151
fscanf() ... 160, 290, 307
 definition of ... 151
 rationale ... 290
fseek() ... 152, 159, 161, 290, 307
 definition of ... 151
 rationale ... 290
F_SETFD ... 122-123, 126, 270, 302
F_SETFL ... 122, 124, 126, 270, 302
F_SETLK ... 122, 124-126, 271, 302
F_SETLKW ... 122, 124-127, 271, 302
fsetpos() ... 214, 272, 307
fstat() ... 21-22, 33, 101, 103-104, 261, 308
 definition of ... 103
 rationale ... 260
ftell() ... 161, 290, 307
 definition of ... 151
 rationale ... 290
 function prototypes ... 32
F_UNLCK ... 122, 124-125, 302
fwrite() ... 160, 272, 290, 307
 definition of ... 151
 rationale ... 290
F_WRLCK ... 122, 124-126, 302

G

General Concepts ... 21, 208
 General File Creation ... 88, 257
 General Terminal Interface Control Functions
 ... 143, 281
 General Terminal Interface ... 129, 274
 General Terms ... 11, 198
 General ... 1
 Generate Terminal Pathname ... 78, 251
 Get and Set State ... 143, 282
 Get Configurable Pathname Variables
 ... 110, 263

- Get Configurable System Variables ... 80, 252
- Get Effective Group ID ... 68, 246
- Get Effective User ID ... 68, 246
- Get File Status ... 103, 260
- Get Foreground Process Group ID ... 147, 282
- Get Password From User ... 252
- Get Process and Parent Process IDs ... 67, 246
- Get Process Group ID ... 72, 247
- Get Process Times ... 76, 250
- Get Real Group ID ... 68, 246
- Get Real User ID ... 68, 246
- Get Supplementary Group IDs ... 70, 247
- Get System Name ... 74
- Get System Time ... 75, 249
- Get User Name ... 71, 247
- Get Working Directory Pathname ... 87
- getc()* ... 160, 290, 307
 - definition of ... 151
 - rationale ... 290
- getchar()* ... 160, 290, 307
 - definition of ... 151
 - rationale ... 290
- getcwd()* ... 86-87, 256, 311
 - definition of ... 87
 - rationale ... 256
- getegid()* ... 68, 247, 311
 - definition of ... 68
 - rationale ... 246
- getenv()* ... 77-78, 189, 229, 307
 - definition of ... 77
 - rationale ... 250
- geteuid()* ... 68, 311
 - definition of ... 68
 - rationale ... 246
- getgid()* ... 68, 311
 - definition of ... 68
 - rationale ... 246
- getgrnt()* ... 293
 - rationale for omission ... 293
- getgrgid()* ... 33, 166, 293, 303
 - definition of ... 166
 - rationale ... 293
- getgrnam()* ... 33, 166, 211, 293, 303
 - definition of ... 166
 - rationale ... 293
- getgroups()* ... 70, 247, 311
 - definition of ... 70
 - rationale ... 247
- gethostid()* ... 249
- gethostname()* ... 249
- getlogin()* ... 71, 166-167, 247, 311
 - definition of ... 71
 - rationale ... 247
- getpass()* ... 252
 - rationale for omission ... 252
- getpgrp()* ... 72, 74, 204, 247, 311
 - definition of ... 72
 - rationale ... 247
- getpid()* ... 57, 67, 73, 241, 311
 - definition of ... 67
 - rationale ... 246
- getppid()* ... 67, 311
 - definition of ... 67
 - rationale ... 246
- getpwent()* ... 293
 - rationale for omission ... 293
- getpwnam()* ... 33, 71, 167, 211, 247, 293, 305
 - definition of ... 167
 - rationale ... 293
- getpwuid()* ... 33, 71, 167, 293, 305
 - definition of ... 167
 - rationale ... 293
- gets()* ... 160, 290, 307
 - definition of ... 151
 - rationale ... 290
- gettimeofday()* ... 249-250
- getty* ... 276
- getuid()* ... 68, 70, 215, 241, 311
 - definition of ... 68
 - rationale ... 246
- getwd()* ... 256
- gid_t* ... 15, 20, 68, 70, 101, 107, 166-167, 214-215, 293, 309
 - definition of ... 27
- GKS ... 196
- gmtime()* ... 152, 200, 310
 - definition of ... 152
- Graphics Standards ... 181
- gr_gid* ... 166, 303
- gr_mem* ... 166, 303
- gr_name* ... 166, 303
- Group Database Access ... 166, 293
- group file ... 201
- group ID
 - definition of ... 15
 - effective ... 13, 68
 - get effective ... 68
 - get real ... 68
 - real ... 18, 68
 - saved set- ... 19
 - set ... 68
 - supplementary ... 20, 70

group
 change file ... 107
group
 Structure ... 166
groups
 multiple (see supplementary group ID)
<grp.h> ... 166, 173, 177, 215, 293, 303
 definition of ... 166

H

handle
 definition of ... 158
harmonized
 definition of ... 314
Header Contents Samples ... 301
header
 definition of ... 29
Headers and Function Prototypes ... 32, 223
hierarchy
 file ... 22
Historical Documentation and Introductory
 Texts ... 182
historical implementations ... 201
HOME
 definition of ... 27
 variable ... 27
hosted implementation ... 201
HUPCL ... 135, 138

I

ICANON ... 134, 139-141
ICRNL ... 134, 136-137
IEXTEN ... 135, 139-140
IFS
 variable ... 217
IGNBRK ... 136
IGNCR ... 134, 136-137
IGNPAR ... 136-137
Implementation Conformance ... 2, 190
implementation defined ... 3-4, 11, 14, 16-18,
 21, 24, 37-38, 42-43, 49-50, 53, 62, 75, 89,
 91, 94-95, 103-104, 106-107, 117, 119, 121,
 127, 129-132, 134-141, 145, 152, 159, 162,
 165, 169, 171, 175-177, 185, 198-199, 201,
 206-209, 211-213, 218, 230, 232, 235-236,
 238, 241, 243-244, 256-259, 272, 284-288,
 291, 294, 296, 299
 definition of ... 10
 rationale ... 197

implementation ... 201
 historical ... 182, 201
 hosted ... 187, 201
 native ... 206
 specific ... 201
incomplete pathname ... 202
init ... 205, 233, 241
INLCR ... 136-137
ino_t ... 101, 309
 definition of ... 27
INPCK ... 136-137
Input and Output Primitives ... 113, 264
Input and Output ... 116, 266
Input Modes ... 136, 279
Input Processing and Reading Data ... 131,
 278
Interactions of Other *FILE*-Type C Functions
 ... 158, 288
interchange
 cpio ... 173
 multiple volumes ... 177
 tar ... 169
Interface Characteristics ... 129, 275
International Standardized Profile (ISP)
 definition of ... 313
inter-process channel ... 113
INTR ... 133-134, 139-140
{INT_MAX} ... 34, 304
{INT_MIN} ... 34, 304
Invariant Value ... 37
Invariant Values ... 37
iocntl() ... 274
ioctl() ... 213, 273-275, 282-283
IPC (see inter-process channel)
IRV
 abbreviation ... 20
 definition of ... 20
isalnum() ... 301
 definition of ... 151
isalpha() ... 301
 definition of ... 151
isascii() ... 188
isatty() ... 79, 188, 212, 311
 definition of ... 79
 rationale ... 251
iscntrl() ... 301
 definition of ... 151
isdigit() ... 301
 definition of ... 151
isgraph() ... 301
 definition of ... 151

ISIG ... 133-134, 139-140
islower() ... 301
 definition of ... 151
 ISO/IEC 10646 ... 182
 ISO/IEC 646 ... 2, 20, 169, 171-174, 176, 207, 295
 rationale for selection ... 295
 ISO/IEC 9899 ... 2, 20
 ISO/IEC 9945-2 ... 181, 226, 228, 286, 289, 294, 315, 318-319
 ISO/IEC Conforming POSIX.1 Application ... 4
isprint() ... 301
 definition of ... 151
 ISP
 definition of ... 313
ispunct() ... 301
 definition of ... 151
isspace() ... 301
 definition of ... 151
 ISTRIP ... 136-137
isupper() ... 301
 definition of ... 151
isxdigit() ... 301
 definition of ... 151
 IXOFF ... 134, 136, 140, 146, 224
 IXON ... 134, 136-137, 140

J

jmp_buf ... 305
 Job Control Signals ... 52
 job control ... 4, 15, 38, 47, 50-52, 73, 118, 121, 129-130, 134-135, 139-140, 143, 197, 202-207, 212, 231, 233-234, 236, 239, 242, 247-249, 251, 273, 275, 277
 definition of ... 15
 implementing applications ... 204
 implementing shells ... 202
 implementing systems ... 205
 set process group ID ... 73

K

kernel ... 206
 kill ... 235
kill() ... 25, 33, 42, 51, 54, 56-57, 60-61, 65, 67, 73, 187, 189, 214, 235-237, 239-242, 283, 306
 definition of ... 56
 rationale ... 241

KILL ... 132, 134, 139
 KornShell ... 216

L

labels
 ANSI ... 300
labs() ... 307
 LANG
 definition of ... 28
 variable ... 28, 155
 Language Standards ... 181
 Language-Dependent Services for the C Programming Language ... 5, 192
 Language-Specific Services for the C Programming Language ... 151, 283
lconv ... 304
 LC_ALL
 definition of ... 28
 variable ... 28, 154-155, 286-287, 304
 LC_*
 variable ... 216
 LC_COLLATE
 definition of ... 28
 variable ... 28, 154, 286, 304
 LC_CTYPE
 definition of ... 28
 variable ... 28, 154, 286, 304
 LC_MESSAGES
 variable ... 320
 LC_MONETARY
 definition of ... 28
 variable ... 28, 154, 286, 304
 LC_NUMERIC
 definition of ... 28
 variable ... 28, 154, 286, 304
 LC_TIME
 definition of ... 28
 variable ... 28, 154, 286, 304, 320
 L_ctermid ... 78, 251
 L_cuserid ... 30, 306
ldexp() ... 305
 definition of ... 151
ldiv() ... 307
ldiv_t ... 307
 library routine ... 206
 {LIMIT} ... 9, 34
 limits ... 34
 C language ... 34
 invariant values ... 37
 minimum values ... 34
 pathname values ... 36
 run-time increasable values ... 34
 run-time invariant values ... 35

<limits.h> ... 3, 34-37, 80, 110-111, 188,
190, 223-225, 236, 251, 263, 304, 318
definition of ... 34

Line Control Functions ... 145, 282

link (see directory entry)

link count
definition of ... 16

Link to a File ... 92, 258

link
definition of ... 15
symbolic ... 294, 297-298

link() ... 24, 92-93, 97, 101, 103, 177, 199,
258-259, 311
definition of ... 92
rationale ... 258

{LINK_MAX} ... 25, 93-94, 100, 111, 304
definition of ... 36

l_len ... 124-125, 303

Local Modes ... 139, 280

Locale Definitions ... 319

locale
C ... 154-155, 286
POSIX ... 154-155, 286

localeconv() ... 304

<locale.h> ... 154, 304

localtime() ... 152-153, 162, 200, 284, 310
definition of ... 152

lockf() ... 271

locking ... 270
advisory ... 124, 272
exclusive ... 271
mandatory ... 272

log10() ... 305
definition of ... 151

log() ... 305
definition of ... 151

logical device ... 206

login name
definition of ... 16

login shell ... 228

login ... 228, 246

login
definition of ... 16

LOGNAME
definition of ... 28
variable ... 28, 216

longjmp() ... 66, 162, 189, 213, 240, 245,
291-292, 305
definition of ... 151
rationale ... 291

_longjmp() ... 291

{LONG_MAX} ... 34, 304

{LONG_MIN} ... 34, 304

l_pid ... 125, 303

lread() ... 214, 266
rationale for omission ... 266

lseek() ... 26, 37, 91, 118, 121, 127-128, 152,
158-161, 193, 214, 272-273, 290, 311
definition of ... 127
rationale ... 272
symbolic constants ... 37

l_start ... 124-125, 303

l_sysid ... 271

l_type ... 126, 303
Values for Record Locking With *fcntl()*
... 122

l_whence ... 124-125, 303

lwrite() ... 266
rationale for omission ... 266

M

machine ... 74, 309

macro
feature test ... 13

MAIL
variable ... 217

main() ... 43, 46, 48, 53, 156, 228, 232
definition of ... 43

Make a Directory ... 94, 258

Make a FIFO Special File ... 95, 259

make ... 233

malloc() ... 188, 240, 256, 291-292, 307
definition of ... 151

Manipulate Signal Sets ... 57, 242

Map a Stream Pointer to a File Descriptor
... 156, 287

Mask for Use With File Access Modes ... 123

mask
file creation ... 91

<math.h> ... 305

{MAX_CANON} ... 111, 132, 304
definition of ... 36

{MAX_CHAR} ... 224

{MAX_INPUT} ... 111, 131-132, 224, 304
definition of ... 36

may
definition of ... 10
rationale ... 197

mblen() ... 307

{MB_CUR_MAX} ... 315

[MB_LEN_MAX] ... 34, 304, 315
mbstowcs() ... 307
mbtowc() ... 307
memchr() ... 308
memcmp() ... 308
memcpy() ... 308
memmove() ... 308
memory management
 omitted functions ... 292
memset() ... 308
Minimum Values ... 34-35, 223
MIN ... 132-133, 139
mkdir() ... 21, 33, 91-92, 94, 99, 103, 107,
 177, 253, 258-259, 308
 definition of ... 94
 rationale ... 258
mkfifo() ... 21, 33, 91-92, 95-96, 103, 107,
 201, 253, 259, 308
 definition of ... 95
 rationale ... 259
mknod() ... 201, 259
mktime() ... 152-153, 162, 200, 310
 definition of ... 152
mode bit ... 44, 172, 257
mode
 definition of ... 16
Modem Disconnect ... 135, 279
mode_t ... 88, 91, 94-95, 101-102, 106, 214,
 258, 260, 309
 definition of ... 27
modf() ... 305
 definition of ... 151
modtime ... 109, 312
mount point ... 206
mount() ... 206
mounted file system ... 206
multibyte character ... 12, 135, 201, 278,
 280, 295
multiple groups (see supplementary group ID)
Multiple Volumes ... 177, 299

N

name
 login ... 16
 system ... 74
 user ... 20
[NAME_MAX] ... 14, 22-23, 25, 39, 46, 83, 85-
 86, 90, 93, 95-98, 100, 104-106, 108-109,
 111-112, 225, 262-263, 304
 definition of ... 36

namespace pollution ... 218
**<National Body> Conforming POSIX.1 Applica-
 tion** ... 5
native implementation ... 206
NDEBUG ... 301
Networking Standards ... 179
new() ... 292
[NGROUPS_MAX] ... 4, 20, 70, 80, 247, 304
nlink_t ... 101, 214, 309
 definition of ... 27
NL ... 134, 137, 139
nodename ... 74, 309
NOFLSH ... 139-140
nohup ... 229
Noncanonical Mode Input Processing ... 132,
 278
Nonlocal Jumps ... 162, 291
Normative References ... 2, 189
null character
 definition of ... 29
null string
 definition of ... 13, 16
NULL ... 304, 306-307, 310-311
 definition of ... 29
Numerical Limits ... 34, 223

O

O_ACCMODE ... 123, 302
O_APPEND ... 88, 119, 122, 161, 288, 302
obsolescent ... 81, 143-144, 197, 252, 265,
 281
 definition of ... 10
 rationale ... 197
O_CREAT ... 88-90, 122, 302
O_EXCL ... 88-90, 122, 302
off_t ... 101, 125, 127, 214, 309
 definition of ... 27
oflag
 Values for *open()* ... 122
Omitted Memory Management ... 292
O_NDELAY ... 258, 264, 269-270
O_NOCTTY ... 89, 122, 130, 257, 302
O_NONBLOCK ... 89-90, 113, 117-120, 122,
 129, 131, 133, 138, 258, 264, 267-270, 302
Open a File ... 88, 257
Open a Stream on a File Descriptor ... 157,
 287
open file description ... 206
 definition of ... 16

open file
 definition of ... 16
Open System Environment (OSE)
 definition of ... 314
open() ... 21, 33, 45, 88-92, 97, 104, 114-116,
 118, 121, 127-131, 137-140, 156-157, 160,
 176, 193, 201, 253, 257-258, 260-261, 265,
 277, 287, 291, 303
 definition of ... 88
 rationale ... 257
opendir() ... 33, 83-85, 176, 254-255, 302
 definition of ... 83
 rationale ... 254
Opening a Terminal Device File ... 129, 275
{OPEN_MAX} ... 25, 80, 84, 114-115, 126, 304
 definition of ... 36
OPOST ... 137
O_RDONLY ... 88-89, 123, 257-258, 302
O_RDWR ... 88-90, 123, 257-258, 302
orphaned process group ... 206, 233, 239
 definition of ... 16
OSE (see Open System Environment)
Other C Language Functions ... 162, 291
Other C Language-Related Specifications
 ... 6, 194
Other Language-Related Specifications ... 7,
 195
Other Standards ... 181
O_TRUNC ... 39, 89-91, 111, 122, 257, 302
Output Modes ... 137, 280
owner
 change file ... 107
O_WRONLY ... 88-90, 123, 258, 302

P

Parameters That Can Be Set ... 135, 279
PARENB ... 138
parent directory ... 207
 definition of ... 16
parent process ID
 definition of ... 16
parent process
 definition of ... 16
PARMRK ... 136-137
PARODD ... 138
passwd file ... 207
passwd ... 167
 Structure ... 167
<passwd.h> ... 215

path prefix
 definition of ... 17
pathconf() ... 37, 39, 110-112, 202, 226, 262-
 263, 311
 definition of ... 110
 rationale ... 263
pathname component
 definition of ... 17
pathname resolution
 definition of ... 22, 211
Pathname Variable Values ... 36, 224
pathname
 absolute ... 11
 definition of ... 17
 get configurable values ... 110
 get working directory ... 87
 incomplete ... 202
 relative ... 19
PATH
 definition of ... 28
 variable ... 28, 43, 46, 216, 230
{PATH_MAX} ... 17, 25, 46, 85-86, 90, 93, 95-
 98, 100, 104-106, 108-109, 111-112, 225,
 252, 256-257, 263, 297-298, 304
 definition of ... 36
pause() ... 49, 63-66, 238, 241, 245, 311
 definition of ... 64
 rationale ... 245
pclose() ... 231
permission
 file ... 15
permissions
 file access ... 21
perror() ... 161, 212, 290, 307
 definition of ... 151
 rationale ... 290
{PID_MAX} ... 214
pid_t ... 18, 41, 47, 56, 67, 72-73, 124, 147-
 148, 214-215, 223, 309
 definition of ... 27
pipe ... 14, 26, 34, 36, 51, 55, 102, 111, 113,
 116-117, 119-121, 128, 202-203, 207, 227,
 238, 257, 265, 267-269, 289, 311
 definition of ... 17
pipe() ... 17, 96, 103-104, 113-116, 118, 121,
 265, 287, 311
 definition of ... 113
 rationale ... 265
{PIPE_BUF} ... 111, 119-120, 267-269, 304
 definition of ... 36
{PIPE_MAX} ... 269
Pipes ... 113, 265

- popen()* ... 231, 289
- portability constants
 - compile-time ... 38
 - execution-time ... 38
- portable filename character set ... 207
 - definition of ... 17
- portable filenames ... 22
- <posix.h> ... 225
- POSIX locale ... 154-155, 286
- POSIX Open System Environment
 - definition of ... 314
- POSIX.1 and the C Standard ... 186
- POSIX.1 Symbols ... 30, 217
- POSIX.1
 - abbreviation ... 21
 - definition of ... 21
- {_POSIX_ARG_MAX} ... 304
 - definition of ... 34
- {_POSIX_CHILD_MAX} ... 304
 - definition of ... 34
- {_POSIX_CHOWN_RESTRICTED} ... 4, 107-108, 111, 263, 311
 - definition of ... 39
- {_POSIX_JOB_CONTROL} ... 4, 15, 73, 80, 129, 147-148, 311
 - definition of ... 38
- {_POSIX_LINK_MAX} ... 304
 - definition of ... 34
- {_POSIX_MAX_CANON} ... 304
 - definition of ... 34
- {_POSIX_MAX_INPUT} ... 304
 - definition of ... 34
- {_POSIX_NAME_MAX} ... 304
 - definition of ... 34
- {_POSIX_NGROUPS_MAX} ... 304
 - definition of ... 34
- {_POSIX_NO_TRUNC} ... 22-23, 25, 46, 85-86, 90, 93, 95-98, 100, 104-106, 108-109, 111-112, 225, 311
 - definition of ... 39
- {_POSIX_OPEN_MAX} ... 304
 - definition of ... 34
- {_POSIX_PATH_MAX} ... 304
 - definition of ... 34
- {_POSIX_PIPE_BUF} ... 304
 - definition of ... 34
- {_POSIX_SAVED_IDS} ... 44, 56, 69-70, 80, 229, 311
 - definition of ... 38
- _POSIX_SOURCE ... 32, 218, 221
 - definition of ... 31
- {_POSIX_SSIZE_MAX} ... 304
 - definition of ... 34
- {_POSIX_STREAM_MAX} ... 304
 - definition of ... 34
- {_POSIX_TZNAME_MAX} ... 304, 318
 - definition of ... 34
- {_POSIX_VDISABLE} ... 111, 135, 141, 311
 - definition of ... 39
- {_POSIX_VERSION} ... 80, 249, 311
 - definition of ... 38
- pow()* ... 305
 - definition of ... 151
- Primitive System Data Types ... 27, 213
- printf()* ... 160-161, 193, 290, 307
 - definition of ... 151
 - rationale ... 290
- privileges (see appropriate privileges)
- Process Creation and Execution ... 41, 226
- Process Creation ... 41, 226
- Process Environment ... 67, 246
- process group ID ... 15, 18, 27, 41, 44, 47, 56, 72-74, 129, 147-149, 203, 247-248, 275-276
 - definition of ... 18
 - foreground ... 15
 - get ... 72
 - get foreground ... 147
 - set ... 72
 - set for job control ... 73
 - set foreground ... 148
- process group leader
 - definition of ... 18
- process group lifetime
 - definition of ... 18
- process group
 - background ... 12
 - definition of ... 17
 - foreground ... 15
 - leader ... 18
 - lifetime ... 18
 - orphaned ... 16, 206, 233, 239
 - terminal ... 129
- Process Groups ... 72, 129, 247, 275
- process groups
 - concepts in job control ... 203
- Process Identification ... 67, 246
- process ID
 - 1 ... 233
 - definition of ... 18
 - get ... 67
 - get parent ... 67
 - parent ... 16
 - rationale ... 214

- process lifetime ... 242
 - definition of ... 18
- Process Primitives ... 41, 226
- Process Termination ... 46, 230
- process
 - background ... 12
 - child ... 12
 - concurrent execution ... 227
 - controlling ... 12
 - create inter-process channel ... 113
 - creation ... 41
 - definition of ... 17
 - delay execution ... 65
 - foreground ... 15
 - parent ... 16
 - send signal to ... 56
 - single-threaded ... 227
 - suspend execution ... 64
 - system ... 20
 - terminate ... 49
 - times ... 76
 - wait for termination ... 47
- profile
 - example ... 318
- Profiles ... 313
- prototypes
 - function ... 32
- PS1
 - variable ... 217
- PS2
 - variable ... 217
- ptrace()* ... 212, 231-232, 243
- ptrdiff_t* ... 306
- putc()* ... 160, 290, 307
 - definition of ... 151
 - rationale ... 290
- putchar()* ... 160, 290, 307
 - definition of ... 151
 - rationale ... 290
- putenv()* ... 250
- puts()* ... 160, 290, 307
 - definition of ... 151
 - rationale ... 290
- pwd* ... 256
- <pwd.h>* ... 27, 167, 173, 177, 215, 293, 305
 - definition of ... 167
- pw_dir* ... 167, 305
- pw_gid* ... 167, 305
- pw_name* ... 167, 305
- pw_shell* ... 167, 305
- pw_uid* ... 167, 305

Q

- qsort()* ... 188, 307
 - definition of ... 151
- QUIT ... 133-134, 139-140

R

- raise()* ... 54, 61, 189, 306
- rand()* ... 307
 - definition of ... 151
- raw mode ... 273
- Read from a File ... 116, 267
- read()* ... 91, 103, 114, 116-119, 128, 130-135, 158, 160, 177, 202, 213, 215, 238-240, 253, 264-267, 269, 277-278, 299-300, 311
 - definition of ... 116
 - rationale ... 267
- readdir()* ... 33, 83-85, 254-255, 302
 - definition of ... 83
 - rationale ... 254
- read-only file system
 - definition of ... 18
- readv()* ... 269
- real group ID ... 15, 44, 68-70, 105
 - definition of ... 18
- real user ID ... 20, 34-35, 44, 68-69, 105, 241, 261
 - definition of ... 18
- realloc()* ... 307
 - definition of ... 151
- Referenced C Language Routines ... 151, 283
- regular file ... 207
 - definition of ... 19
- Related Activities ... 315
- Related Functions by Both ... 189
- Related Open Systems Standards ... 179
- Related Standards ... 315
- Relationship to IEEE Draft Project 1003.0 ... 315
- relative pathname
 - definition of ... 19
- release* ... 74, 309
- Remove a Directory ... 98, 259
- Remove Directory Entries ... 96, 259
- remove()* ... 103, 162, 307
 - definition of ... 151
- Rename a File ... 99, 259
- rename()* ... 93, 97, 99-100, 151, 189, 259, 307
 - definition of ... 99
 - rationale ... 259

Reposition Read/Write File Offset ... 127, 272
 Required Signals ... 52
 Requirements ... 2, 190
 Reserved Header Symbols ... 31
rewind() ... 161, 290, 307
 definition of ... 151
 rationale ... 290
rewinddir() ... 33, 83-84, 255, 302
 definition of ... 83
 rationale ... 254
rmdir() ... 97-98, 101, 213, 259, 311
 definition of ... 98
 rationale ... 259
 R_OK ... 37-38, 105, 311
 root directory ... 207
 definition of ... 19
 root file system ... 208
 root of a file system ... 208
 Run-Time Increaseable Values ... 34-35, 224
 Run-Time Invariant Values (Possibly Indeterminate) ... 35-36, 224

S

sa_flags ... 59, 306
sa_handler ... 59, 306
sa_mask ... 59, 243, 306
samefile() ... 214
 Sample National Profile ... 317
 SA_NOCLDSTOP ... 53, 59, 203, 243, 305
 saved set-group-ID ... 15, 38, 44, 69-70, 246
 definition of ... 19
 saved set-user-ID ... 19-20, 38, 44, 56, 69, 246
 definition of ... 19
sbrk() ... 188, 292
 rationale for omission ... 292
scanf() ... 160, 193, 290, 307
 definition of ... 151
 rationale ... 290
 {SCHAR_MAX} ... 34, 304
 {SCHAR_MIN} ... 34, 304
 Schedule Alarm ... 63, 244
 Scope and Normative References ... 185
 Scope ... 1, 185
 seconds since the Epoch ... 76, 103, 109, 152, 200, 208, 284
 definition of ... 19
 security considerations ... 4, 21, 56, 199, 201, 205, 208-209, 216, 230, 242, 248, 252, 260, 262, 277, 282-283, 293, 295-296

security controls
 additional ... 21
 alternate ... 21
 extended ... 21
 security
 access control lists ... 209
 mandatory ... 209
 monolithic privileges ... 199
seek [see *lseek()*]
seekdir() ... 255-256
 rationale for omission ... 255
 SEEK_CUR ... 30, 38, 124, 127, 303, 311
 SEEK_END ... 30, 38, 124, 127, 303, 311
 SEEK_SET ... 30, 38, 124-125, 127, 289, 303, 311
select() ... 266
 Send a Signal to a Process ... 56, 241
 session leader
 definition of ... 19
 session lifetime
 definition of ... 19
 session ... 12, 15-16, 19, 44, 50, 56, 72-74, 130, 148-149, 179, 204, 206-207, 233, 242, 248, 276, 283
 create ... 72
 definition of ... 19
 leader ... 19
 lifetime ... 19
 Set File Access and Modification Times ... 108, 262
 Set File Creation Mask ... 91, 258
 Set Foreground Process Group ID ... 148, 283
 Set Position of Directory Stream ... 255
 Set Process Group ID for Job Control ... 73, 248
 Set Time Zone ... 162, 292
 Set User and Group IDs ... 68, 246
setbuf() ... 307
 definition of ... 151
setcooked() ... 273
setgid() ... 68-71, 311
 definition of ... 68
 rationale ... 246
setgrent() ... 293
 rationale for omission ... 293
 set-group-ID ... 44, 107, 229-230, 272
setgroups() ... 247
sethostid() ... 249
sethostname() ... 249
setjmp() ... 162, 189, 245, 291-292, 305
 definition of ... 151
 rationale ... 291

- `<setjmp.h>` ... 162, 305
- `_setjmp()` ... 291
- `setlocale()` ... 154-155, 189, 285-287, 304
 - definition of ... 154
 - rationale ... 285
- `setpgid()` ... 18-19, 72-74, 148, 197, 203-204, 233, 248-249, 276, 311
 - definition of ... 73
 - rationale ... 248
- `setpgrp()` ... 248
- `setpwent()` ... 293
 - rationale for omission ... 293
- `setraw()` ... 273
- `setsid()` ... 18, 57, 72-74, 130, 148, 233, 248, 276, 311
 - definition of ... 72
 - rationale ... 248
- `setuid()` ... 44, 46, 68-69, 311
 - definition of ... 68
 - rationale ... 246
- `set-user-ID` ... 44, 107, 229-230, 241, 256
- `setvbuf()` ... 288, 307
- `sh` ... 287
- `shall`
 - definition of ... 10
 - rationale ... 197
- shell scripts
 - execing ... 228
- shell ... 1, 202-205, 216-217, 226, 228, 231, 233, 236, 238-239, 242, 247-249, 275, 277, 288
 - job control ... 202, 236, 239, 242
 - login ... 205, 228, 247
- SHELL**
 - variable ... 217
- `should`
 - definition of ... 10
 - rationale ... 197
- `{SHRT_MAX}` ... 34, 304
- `{SHRT_MIN}` ... 34, 304
- `SIGABRT` ... 46, 51, 235, 291, 305
- `sigaction` ... 58, 306
- `sigaction()` ... 33, 50, 53, 57-60, 62-64, 66, 72-73, 128, 162, 236-237, 241-244, 283, 291, 306
 - definition of ... 58
 - rationale ... 243
- `sigaddset()` ... 33, 57-58, 306
 - definition of ... 57
 - rationale ... 242
- `SIGALRM` ... 51, 64-66, 245, 305
- `sig_atomic_t` ... 306
- `SIGBUS` ... 235-236
- `SIGCHLD` ... 50-51, 53-54, 59, 203, 236-239, 243, 305
- `SIGCLD` ... 235-236, 238, 243
- `SIGCONT` ... 50, 52-53, 55-56, 203, 233-234, 236, 238-239, 242, 305
- `sigdelset()` ... 33, 57-58, 306
 - definition of ... 57
 - rationale ... 242
- `sigemptyset()` ... 33, 57, 242-243, 306
 - definition of ... 57
 - rationale ... 242
- `SIGEMT` ... 235-236
- `sigfillset()` ... 33, 57, 242-243, 306
 - definition of ... 57
 - rationale ... 242
- `SIGFPE` ... 51, 54, 61, 235-236, 238, 305
- `SIGHUP` ... 50-51, 135, 233-234, 239, 305
- `SIGILL` ... 51, 54, 61, 235-236
- `SIGINT` ... 24, 51, 133, 136, 205, 226, 235, 305
- `SIGIOT` ... 235-236
- `sigismember()` ... 33, 57-58, 306
 - definition of ... 57
 - rationale ... 242
- `sigjmp_buf` ... 305
- `SIGKILL` ... 51, 53-54, 59, 61, 235, 237-239, 241, 243, 305
- `siglongjmp()` ... 33, 66, 162, 189, 213, 240, 291, 305
 - definition of ... 162
 - rationale ... 291
- Signal Actions ... 53, 238
- Signal Concepts ... 51, 235
- Signal Effects on Other Functions ... 55, 240
- Signal Generation and Delivery ... 51, 237
- Signal Names ... 51, 235
- `signal` ... 208
 - BSD differences ... 234
 - concepts ... 51
 - definition of ... 19
 - examine and change action ... 58
 - examine and change blocked ... 60
 - examine pending ... 62
 - manipulate sets ... 57
 - mask ... 60
 - queueing ... 235
 - send to process ... 56
 - stacks ... 234
 - wait ... 62
- `signal()` ... 25, 60, 189, 234-235, 237, 241-244, 283, 306
- `<signal.h>` ... 24, 46, 49, 51, 56-64, 91, 162, 305
 - definition of ... 51

- Signals ... 51, 234
- sigpending()* ... 33, 46, 58, 62-63, 234, 242, 306
 - definition of ... 62
 - rationale ... 244
- SIGPIPE ... 51, 121, 213, 305
- sigprocmask()* ... 33, 46, 53, 58-63, 162, 237, 242, 244-245, 291, 306
 - definition of ... 60
 - rationale ... 244
- SIGQUIT ... 24, 51, 133, 305
- sigreturn()* ... 234
- SIG_BLOCK ... 61, 305
- SIG_DFL ... 44, 51, 53-54, 59-60, 205, 234-235, 237-238, 305
- SIG_ERR ... 305
- SIG_IGN ... 44, 51, 53-54, 59, 203, 205, 229, 237-238, 305
- SIG_SETMASK ... 61, 305
- SIG_UNBLOCK ... 61, 305
- SIGSEGV ... 51, 54, 61, 235-236, 305
- sigsetjmp()* ... 33, 162, 189, 291, 305
 - definition of ... 162
 - rationale ... 291
- sigset_t* ... 51, 57, 59-60, 62, 235, 306
- sigstack()* ... 234
- SIGSTOP ... 52-54, 59, 61, 236, 239, 243, 305
- sigsuspend()* ... 33, 53, 58-60, 62-63, 162, 238, 241, 244-245, 291, 306
 - definition of ... 62
 - rationale ... 244
- SIGSYS ... 235-236
- SIGTERM ... 52, 235, 305
- SIGTRAP ... 235-236
- SIGTSTP ... 51, 53, 134, 205, 236, 239, 305
- SIGTTIN ... 51, 53, 118, 130, 204-205, 236, 239, 305
- SIGTTOU ... 51, 53, 121, 131, 140, 143, 203, 205, 236, 239, 277, 282, 305
- SIGUSR1 ... 52, 235-236, 305
- SIGUSR2 ... 52, 235-236, 305
- sigvec()* ... 243
- sin()* ... 305
 - definition of ... 151
- sinh()* ... 305
 - definition of ... 151
- S_IRGRP ... 30, 103, 303, 308
- S_IROTH ... 30, 103, 303, 308
- S_IRUSR ... 30, 103, 303, 308
- S_IRWXG ... 30, 103, 303, 308
- S_IRWXO ... 30, 103, 303, 308
- S_IRWXU ... 30, 103, 303, 308
- S_ISBLK ... 30, 102, 303, 308
- S_ISCHR ... 30, 102, 303, 308
- S_ISDIR ... 30, 102, 303, 308
- S_ISFIFO ... 30, 102, 303, 308
- S_ISGID ... 30, 103, 106-107, 260-262, 303, 308
- S_ISREG ... 30, 102, 303, 308
- S_ISUID ... 30, 103, 106-107, 260-262, 303, 308
- S_IWGRP ... 30, 103, 303, 308
- S_IWOTH ... 30, 103, 155, 303, 308
- S_IWUSR ... 30, 103, 303, 308
- S_IXGRP ... 30, 103, 303, 308
- S_IXOTH ... 30, 103, 303, 308
- S_IXUSR ... 30, 103, 303, 308
- size_t* ... 87, 116, 215, 266, 306-307, 309-311
 - definition of ... 27
- slash
 - definition of ... 20
- sleep()* ... 63, 65-66, 240-241, 244-246, 311
 - definition of ... 65
 - rationale ... 245
- socket()* ... 287
- sockets ... 299
- Solely by POSIX.1 ... 188
- Solely by the C Standard ... 188
- Special Characters ... 133, 279
- Special Control Characters ... 140, 280
- Special File Creation ... 94, 258
- Special Symbol {CLK_TCK} ... 81, 252
- specific implementation ... 201
- speed_t* ... 141, 310
 - definition of ... 142
- sprintf()* ... 307
 - definition of ... 151
- sqrt()* ... 305
 - definition of ... 151
- srand()* ... 307
 - definition of ... 151
- sscanf()* ... 307
 - definition of ... 151
- {SSIZE_MAX} ... 27, 117, 119, 215, 304
 - definition of ... 37
- ssize_t* ... 33-34, 37, 116, 118, 215, 309, 311
 - definition of ... 27
- standards
 - database ... 181
 - graphics ... 181
 - language ... 181
 - networking ... 179
 - open systems ... 179

START ... 134, 137, 141, 146
stat ... 103, 308
 Structure ... 101
stat() ... 21-22, 33, 46, 91, 95-96, 101, 103-105, 107, 172-173, 176-177, 193, 199, 253, 260, 262, 308
 definition of ... 103
 rationale ... 260
 <stat.h>
 (see <sys/stat.h>)
st_atime ... 22, 45, 84, 89, 94-95, 101, 103, 113, 117, 160, 211, 254, 262, 267, 288, 308
 definition of ... 101
st_ctime ... 22, 89-90, 92, 94-95, 97-98, 100-101, 103, 106-107, 109, 113, 120, 160-161, 288, 308
 definition of ... 101
 <stdarg.h> ... 306
 <stddef.h> ... 306
 STDERR_FILENO ... 156, 311
st_dev ... 101, 260, 308
 definition of ... 101
 STDIN_FILENO ... 156, 311
 <stdio.h> ... 30, 78, 156-157, 306
 <stdlib.h> ... 77, 194-195, 307
 STDOUT_FILENO ... 156, 311
st_gid ... 101, 308
 definition of ... 101
st_ino ... 101, 260, 308
 definition of ... 101
st_mode ... 101-102
 definition of ... 101
st_mtime ... 22, 89-90, 92, 94-95, 97-98, 100-101, 103, 113, 120, 160-161, 262, 288, 308
 definition of ... 101
st_nlink ... 101, 214, 308
 definition of ... 101
 STOP ... 134, 137, 141, 146
strcat() ... 308
 definition of ... 151
strchr() ... 308
 definition of ... 151
strcmp() ... 308
 definition of ... 151
strcoll() ... 308
strcpy() ... 308
 definition of ... 151
strcspn() ... 308
 definition of ... 151
st_rdev ... 219
 stream
 binary ... 152
 open ... 157
 pointer ... 156
 text ... 152
 (STREAM_MAX) ... 80, 304
 definition of ... 35
 streams
 Eighth Edition ... 198
strerror() ... 308
strftime() ... 152-153, 162, 310
 definition of ... 152
 Strictly Conforming POSIX.1 Application
 ... 4, 192
 string
 definition of ... 29
 <string.h> ... 307
strlen() ... 308
 definition of ... 151
strncat() ... 308
 definition of ... 151
strncmp() ... 308
 definition of ... 151
strncpy() ... 308
 definition of ... 151
strpbrk() ... 308
 definition of ... 151
strrchr() ... 308
 definition of ... 151
strspn() ... 308
 definition of ... 151
strstr() ... 308
 definition of ... 151
strtod() ... 307
strtok() ... 308
 definition of ... 151
strtol() ... 307
strtoul() ... 307
 struct
 direct ... 253
 dirent ... 83, 85, 253-254, 302
 flock ... 124, 302
 group ... 166, 303
 iconv ... 304
 passwd ... 167, 305
 sigaction ... 58, 306
 stat ... 103, 308
 termios ... 141, 143, 310
 tm ... 310
 tms ... 76, 308
 utimbuf ... 108, 311
 utsname ... 74, 309
 structures
 additions to ... 218
strxfrm() ... 308
st_size ... 308
 definition of ... 101

st_uid ... 101, 308
 definition of ... 101
su ... 242, 246
subshells ... 204
 Suggested Feature Test Macros ... 222
super-user ... 199, 208, 241, 246, 258-259, 261-262, 295
supplementary group ID
 definition of ... 20
supplementary groups ... 14-15, 20, 34, 39, 44-45, 69-70, 106-107, 208, 247, 262
supported
 definition of ... 11
 rationale ... 197
 Suspend Process Execution ... 64, 245
 SUSP ... 134, 139-140
 Symbolic Constant for the *lseek()* Function ... 37
 Symbolic Constants for the *access()* Function ... 37-38, 225
 Symbolic Constants for the *lseek()* Function ... 38, 225
 Symbolic Constants ... 37, 225
symbolic link ... 294, 297-298
 Symbols From the C Standard ... 29, 217
symbols
 C Standard ... 29
 POSIX.1 ... 30
sysconf() ... 34-35, 77, 80-81, 202, 236, 252, 263, 311
 definition of ... 80
 rationale ... 252
 <sys/dir.h> ... 253
sysname ... 74, 309
 <sys/stat.h> ... 46, 88, 91-92, 94-96, 101, 103-104, 106-108, 110, 173, 177, 214-215, 257, 260, 308
 definition of ... 101
 File Modes ... 102
 File Types ... 102
 Time Entries ... 103
system call ... 208
 restarting ... 234
 System Databases ... 165, 293
system documentation
 definition of ... 11
 System Identification ... 74, 249
 System III ... 207, 213, 222, 234, 249, 262, 264, 267, 273
 System Name ... 249
system process
 definition of ... 20

System V ... 199, 205-206, 222, 225, 230, 233-238, 241, 243-244, 246-249, 253, 258-259, 262-265, 267, 270-271, 273-274, 282-283, 291, 297
system
 definition of ... 20
 get configurable variables ... 80
 name ... 74
system() ... 231, 288-289, 307
 <sys/times.h> ... 76, 308
 definition of ... 76
 <sys/types.h> ... 27, 41, 47, 56, 67-68, 70, 72-73, 83, 88, 91, 94-95, 103, 106-108, 121, 127, 147-148, 166-167, 213, 215, 217, 221, 250, 260, 293, 309
 definition of ... 27
 <sys/utname.h> ... 74, 309
 definition of ... 74
 <sys/wait.h> ... 47-48, 309
 definition of ... 47

T

tan() ... 305
 definition of ... 151
tanh() ... 305
 definition of ... 151
tapecntl() ... 274
tar ... 169, 294-297
 format ... 169
 Header Block ... 170
 <tar.h> ... 170
*tc**() ... 275
tcdrain() ... 33, 145-146, 279, 310
 definition of ... 145
 rationale ... 282
tcflag_t ... 310
 definition of ... 136
tcflow() ... 33, 145-147, 310
 definition of ... 145
 rationale ... 282
tcflush() ... 33, 145-146, 310
 definition of ... 145
 rationale ... 282
tcgetattr() ... 33, 39, 141, 143-145, 204, 281-282, 310
 definition of ... 143
 rationale ... 282
tcgetpgrp() ... 147-148, 204, 276, 282, 311
 definition of ... 147
 rationale ... 282
tcsendbreak() ... 33, 145-146, 310
 definition of ... 145
 rationale ... 282

- tcsetattr()* ... 33, 39, 141-145, 204, 275, 281-282, 310
 - definition of ... 143
 - rationale ... 282
- tcsetpgrp()* ... 74, 130, 148-149, 203-204, 283, 311
 - definition of ... 148
 - rationale ... 283
- telldir()* ... 255-256
 - rationale for omission ... 255
- termcntl()* ... 274
- Terminal Access Control ... 130, 277
- terminal device (see terminal)
- Terminal Identification ... 78, 251
- terminal
 - access control ... 130
 - baud rates ... 141
 - canonical mode input ... 132
 - closing ... 135
 - control functions ... 143
 - control modes ... 138
 - controlling ... 12, 130
 - definition of ... 20
 - determine device name ... 79
 - general interface ... 129
 - generate pathname ... 78
 - get and set state ... 143
 - input ... 131
 - input modes ... 136
 - line control functions ... 145
 - local modes ... 139
 - modem disconnect ... 135
 - noncanonical mode input ... 132
 - opening ... 129
 - output ... 133
 - output modes ... 137
 - process group ... 129
 - setting parameters ... 135
 - special characters ... 133
 - special control characters ... 140
- Terminate a Process ... 49, 232
- Terminology and General Requirements ... 9
- Terminology ... 10, 197
- termios c_cc* Special Control Characters ... 140
- termios c_cflag* Field ... 138
- termios c_iflag* Field ... 136
- termios c_lflag* Field ... 139
- termios* ... 141, 143, 310
 - Baud Rate Values ... 141
 - definition of ... 129
 - get and set state ... 143
 - line control functions ... 145
 - setting parameters ... 135
 - Structure ... 135-136, 279
- <termios.h>* ... 135-147, 309
 - definition of ... 135
- TERM**
 - definition of ... 28
 - variable ... 28
- terms ... 11
- text stream ... 152
- time zone ... 29, 152-153, 162, 189, 283-285
- Time ... 75, 249
- time
 - extensions to C Standard functions ... 152
 - get process ... 76
- time()* ... 33, 75-77, 189, 212, 249-250, 310
 - definition of ... 75
 - rationale ... 249
- <time.h>* ... 75, 81, 152, 162, 217, 250, 310
- Timer Operations ... 63, 244
- TIME ... 132-133, 139
- times()* ... 33, 42, 46, 49, 76-77, 199, 212, 244, 249, 308
 - definition of ... 76
 - rationale ... 250
- <times.h>*
 - (see *<sys/times.h>*)
- time_t* ... 29, 75-76, 101, 109, 200, 217, 249-250, 310
 - definition of ... 27
- tm* ... 310
- tm_hour* ... 19, 310
- tm_isdst* ... 310
- tm_mday* ... 200, 310
- tm_min* ... 19, 310
- tm_mon* ... 200, 310
- tmpfile()* ... 161, 290, 307
 - definition of ... 151
 - rationale ... 290
- tmpnam()* ... 307
 - definition of ... 151
- tms* ... 76, 308
- tms_cstime* ... 41, 45, 76-77, 308
- tms_cutime* ... 41, 45, 76-77, 308
- tm_sec* ... 19, 310
- tms_stime* ... 45, 76-77, 308
- tms_utime* ... 45, 76, 308
- tm_wday* ... 310
- tm_yday* ... 19, 200, 310
- tm_year* ... 19, 200, 310
- toascii()* ... 188
- tolower()* ... 301
 - definition of ... 151

_tolower() ... 188
 TOSTOP ... 131, 139-140, 203
toupper() ... 301
 definition of ... 151
_toupper() ... 188
 TRAILER!!! ... 176
 trojan horse ... 199
ttyname() ... 79, 188, 251, 311
 definition of ... 79
 rationale ... 251
 Types of Conformance ... 5, 194
 <types.h>
 (see <sys/types.h>)
tzname ... 153
 {TZNAME_MAX} ... 80, 152, 304
 definition of ... 35
TZ
 definition of ... 29
 rationale ... 283
 variable ... 29, 34-35, 152-153, 162, 216,
 283-285, 319
tzset() ... 33, 153, 162-163, 310
 definition of ... 162

U

{UCHAR_MAX} ... 34, 304
 {UID_MAX} ... 215
uid_t ... 68, 101, 107, 167, 214-215, 293, 309
 definition of ... 27
 {UINT_MAX} ... 34, 245-246, 304
 {ULONG_MAX} ... 34, 304
umask() ... 33, 46, 91-92, 95-96, 258, 308
 definition of ... 91
 rationale ... 258
umount() ... 206
uname() ... 33, 74-75, 249, 309
 definition of ... 74
 rationale ... 249
 Structure Members ... 75
 undefined ... 3, 6-7, 11, 19, 25, 27, 29-30, 32,
 49, 54-55, 58, 61, 70, 84, 87-89, 117, 119,
 124, 127, 132, 135, 137, 144, 153, 158-159,
 170, 176, 190, 193, 197-198, 212, 232, 240,
 251, 257, 288, 298-299, 319
 definition of ... 11
 rationale ... 198
ungetc() ... 160, 307
 definition of ... 151
 <unistd.h> ... 3, 27, 29-30, 32, 37-39, 80,
 104-105, 110-111, 121, 127-128, 156, 173,
 190, 215, 222-223, 225, 236, 251, 311
 definition of ... 37

unlink() ... 93, 96-97, 99, 101, 103, 116, 162,
 213, 258-259, 311
 definition of ... 96
 rationale ... 259
 unsafe functions ... 240
 unspecified ... 3-4, 11-12, 16, 20, 22-24, 27,
 32, 42, 45, 47, 53-54, 56, 60, 62, 65-66, 71,
 75-79, 83-84, 88-89, 92, 98, 101, 111, 116,
 118, 121, 124, 129-130, 135, 141-142, 152,
 154, 156-157, 159, 161, 165-167, 169, 172-
 174, 176, 185, 190, 193, 197-198, 211, 233,
 237, 243-245, 250, 253, 276
 definition of ... 11
 rationale ... 198
 User Database Access ... 167, 293
 User Identification ... 68, 246
 user ID
 definition of ... 20
 effective ... 13, 68
 get effective ... 68
 get real ... 68
 real ... 18, 68
 saved set- ... 19
 set ... 68
 user name
 definition of ... 20
 get ... 71
USER
 variable ... 216
ushort_t ... 213
 {USHRT_MAX} ... 34, 304
ustat() ... 260
utimbuf ... 108, 311
utime() ... 33, 103, 108-109, 262, 312
 definition of ... 108
 rationale ... 262
 <utime.h> ... 108-109, 311
utsname ... 74, 309
 <utsname.h>
 (see <sys/utsname.h>)

V

Values for *cpio c_mode* Field ... 175
 VEOF ... 140
 VEOL ... 140
 VERASE ... 140
 Version 7 ... 211, 222, 234, 241, 244, 249-
 250, 253, 262, 265, 267, 273, 296
version ... 74, 309
vfork() ... 248

ISO/IEC 9945-1: 1990
IEEE Std 1003.1-1990

vfprintf() ... 160-161, 290, 307
 rationale ... 290
vhangup() ... 206, 233
VINTR ... 140
VKILL ... 140
VMIN ... 140-141
vprintf() ... 160-161, 290, 307
 rationale ... 290
VQUIT ... 140
vsprintf() ... 307
VSTART ... 140-141
VSTOP ... 140-141
VSUSP ... 140
VTIME ... 140-141

W

Wait for a Signal ... 62, 244
Wait for Process Termination ... 47, 231
wait3() ... 231
wait() ... 18, 24, 33, 46-50, 76-77, 212, 226,
 231-232, 236, 238, 240-241, 288, 309
 rationale ... 230
<wait.h>
 (see <sys/wait.h>)
waitpid() ... 18, 24, 33, 46-50, 76, 203, 207,
 214, 226, 231-232, 240, 242-243, 309
 definition of ... 47
 rationale ... 230
wchar_t ... 307
wcstombs() ... 307
wctomb() ... 307
WERASE ... 278
WEXITSTATUS ... 48, 309
WIFEXITED ... 48, 309
WIFSIGNALED ... 48, 309
WIFSTOPPED ... 48, 232, 309
WNOHANG ... 47, 243, 309
W_OK ... 37-38, 105, 311
{WORD_BIT} ... 223
Working Directory Pathname ... 256
Working Directory ... 86, 256
working directory
 change ... 86
 definition of ... 20
Write to a File ... 118, 267
write() ... 26, 91, 103, 114, 118-120, 128, 131,
 133, 135, 158, 160-161, 170, 202-203, 213,
 215, 238-240, 264-266, 269, 272, 277, 290,
 300, 311
 definition of ... 118
 rationale ... 267

writew() ... 269
Writing Data and Output Processing ... 133,
 279
WSTOPSIG ... 48, 309
WTERMSIG ... 48, 309
WUNTRACED ... 47, 203, 231-232, 309

X

X_OK ... 37-38, 105, 261, 311

Acknowledgments

We wish to thank the following organizations for donating significant computer, printing, and editing resources to the production of the 1988 version of this standard, upon which this revision is primarily based: UniForum (formerly /usr/group), Amdahl Corporation, Digital Equipment Corporation, MASSCOMP, UniSoft Corporation, and X/Open.

We wish to thank the following organizations for donating significant computer, printing, and editing resources to the production of this revision: Hewlett-Packard Company and X/Open.

This document was also approved by ISO/IEC JTC-1 SC22/WG15 as ISO/IEC 9945-1:1990. The IEEE wishes to thank the advisory groups of the National Bodies participating in WG15 for their contributions: Austria, Belgium, Canada, Denmark, France, Germany, Japan, Netherlands, United Kingdom, USA, and USSR.

The IEEE also wishes to thank the delegates to WG15 for their contributions:

AUSTRIA
Gerhard Schmitt
Wolfgang Schwabl

CANADA
Joe Cote
Patrick Dempster
George Kriger
Bernard Martineau
Major Douglas J. Moore
Arnie Powell
Paul Renaud
Richard Sniderman

CEC
Phil Bertrand
Manuel Carbajo
Michel Colin

DENMARK
Peter E. Cordsen
Isak Korn
Keld Simonsen
Claus Tondering

FINLAND
Jikka Haikala

FRANCE
Pascal Beyles
Christophe Binot
Claude Bourstin
Jean-Michel Cornu

Yves Delarue
Eric Dumas
Maurice Fathi
Gerald Krummeck
Herve Schauer
Hubert Zimmerman

GERMANY
Ron Elliot
Helmut Stiegler
Claus Unger
Rainer Zimmer

IRELAND
Hans-Jurgen Kugler

JAPAN
Hiromichi Kogure
Shigekatsu Makao
Yasushi Nakahara
Nobuo Saito

NETHERLANDS
J. Van Katwijk
Willem Wakker
H.J. Weegenaar

SWEDEN
Mat Linder

UK
Nigel Bevan
Cornelia Boldyreff
Dave Cannon
Don Chacon
Dominic Dunlop
David Flint
Don E. Follond
Martin Kirk
Neil Martin
Brian Meek
Kevin Murphy
Ian Newman
Philip Rushton

USA
Robert Bismuth
Steven L. Carter
Terence S. Dowling
Ron Elliott
Dale Harris
John Hill
James D. Isaak
Hal Jespersen
Roger J. Martin
Shane McCarron
Barry Needham
Donn S. Terry
Alan Weaver

USSR
V. Koukhar
Ostapenko Georgy Pavlovich

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.

Absolut Software	Bull Sems
ACM	Burroughs Corporation
ADDAMAX	Burtek, Inc.
Adobe Systems, Inc.	C. S. Draper Lab, Inc.
Aeon Technologies, Inc.	California State University
AFNOR*	California University
AFUU	Calspan Corporation
AGS Information Services	Carnegie-Mellon University
Air Force Institute of Technology	CCTA, Riverwalk House
Aktiengesellschaft	Celerity Computing
Alcyon Corporation	Central Computer & Telecommunications
Alliant Computer Systems	Central Intelligence Agency
Alsup	Centre National D'Etudes des Telecomm.
Amdahl Corporation*	CFI
Analysts International	Charles River Data Systems*
ANFOR	Chemical Abstracts Service
Anistics, Inc.	Chorus Systems
ANSI	Citicorp Transaction Tech., Inc.
Applicon	Classic Conferences
Applied Network Technology	Cleveland State University
APT Data Services	CMC Ltd.
Archives Ltd.	Commission of the European Communities**
ARIX	CommUNIXations
Associated Computer Experts	COMPASS
AT Computing/Toernooiveld	Compugraphic Corporation
Atlanta UNIX Users Group	Computer Design
AT&T*	Computer Systems Engineering
AT&T Bell Laboratories	Computer Systems News
AT&T Information Systems	Computer Systems Resources, Inc.
AT&T UNIX Europe Ltd.	Computer Task Group
ATTAGE	Computer Works
Australian Government Dept. of Science	Computer X, Inc.
Australian UNIX Systems User Group	Computerworld
Automation Resource Group	Comtek
Automation Technologies	Concord Data Systems
Axon Data	Concurrent Computer Corporation*
Baldwin Information Processing	Control Data Corporation*
Barrister Information Systems Corporation	Convergent Technologies
Batelle Columbus Labs	Convex Computer Corporation
BBN Communications Corporation	COSMIC
BBN Laboratories	Cray Research, Inc.*
Bell Communications Research	Cullinet Software, Inc.
Bell-Northern Research Ltd.	Custom Development Environments
Billy Shakespeare & Company	Dana Computers
Boeing Aerospace Company	Dansk Data Elektronik A/S DDE
Boeing Computer Services	Dansk Standardiseringsradd
BP America Research & Development	Dartmouth College
Brake Systems, Inc.	Data Connection
British Airways	Data General Corporation
British Olivetti Ltd.	Data Logic Ltd.
British Standards Institute*	Data Systems Engineering
British Telecom	DataBoard, Inc.
Brown & Sharpe	Datamation
Bull, Inc.	Datamension Corporation

Datapoint Corporation
 DEC International
 Defense Communication Agency
 Deutsches Institut für Normung
 DGM&S, Inc.
 Digital Equipment Corporation**
 Digital Equipment GmbH
 Digital Sound Corporation
 Directorate Land Armament
 Douglas Aircraft Corporation
 Dravo Automation Sciences (DAS)
 Eastman-Stuart Ltd.
 Eclipse Systems Corporation
 EDS Corporation
 EDS of Canada, Ltd.
 EDV Zentrum der TU Wien
 EER Systems Corporation
 Electronic Data Systems Corporation
 Electronic Engineering Times
 Electrospace Systems, Inc.
 Emerald City
 Emerging Technologies Group, Inc.
 Encore Computer
 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.
 Federal Judicial Center
 Fern Universität
 FGAN/FFM
 Fidcom System Ltd.
 Flexible Automation
 Flexible Computer Corporation
 Floating Point Systems
 Ford Aerospace Corporation
 Ford Motor Company
 Fortune Systems Corporation
 Fourth Shift 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 Corporation
 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*
 Hayden Publishing
 HCR Corporation
 Hewlett-Packard Company**
 Hewlett-Packard/Apollo*
 High Tech Publishing Company, Inc.
 Hi-Tech Editorial, Inc.
 Honeywell Bull, Inc.
 Honeywell ISI
 Honeywell Ltd.
 Hughes Aircraft Company
 Human Computing Resources
 IAI MBT
 IBM Corporation**
 IBM Deutschland
 IBM Federal Sector 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
 Integrated Systems Design, Inc.
 Intel Corporation
 Interactive Systems Corporation
 Intergraph Corporation
 International Bureau of Software Test
 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
 ITSCJ*
 ITT
 ITUSA

Japanese Industrial Standards Committee
 Johnson Controls, Inc.
 KAIST
 Kendall Square Research Corporation
 K.E.T.R.I.
 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
 Maharishi International University
 Mallinckrodt Institute of Radiology
 Mark Williams Company
 Markor
 Martin Marietta Aerospace
 Martin Marietta Data Systems
 Martin Marietta Energy Systems
 MASSCOMP
 Maxim Technologies, Inc.
 M.B.F. Systems, Inc.
 McDonnell Douglas
 McDonnell Douglas Computer Company
 McGill University
 Mercury Computer Systems, Inc.
 Meridian Software Systems
 Microfocus
 Micrology, Inc.
 Microsoft Corporation
 Microtel Pacific Research
 Microtex
 Mindcraft, Inc.
 Mini-Micro Systems
 MIPS Computer Systems
 MIT-LCS
 Mitsubishi Electric Corporation
 Modcomp*
 Modular Systems
 Molecular Genetics, Inc.
 Monarch Data Systems
 Mortice Kern Systems, Inc.
 Motorola, Inc.
 Motorola Israel, Inc.
 Motorola MCD
 MT XINU
 NAPS, Inc.
 NASA
 NASA Ames Research Center
 NASA Johnson Space Center
 NASA Kennedy Space Center
 National Cancer Institute

National Computer Security Center
 National Electrical Manufacturers Assoc.
 National Institute of Standards & Technology
 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
 Open Software Foundation*
 Open Systems Architects, Inc.
 Oren Yuen Associates
 Osterreichisches Normungsinstitut
 Oxford Systems, Inc.
 Pacific Marine Technology
 Palladium Data Systems
 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
 Politecnico Di Torino-DIP Automatic
 Politecnico Di Mikeno
 Portland Community College
 POSIX Software Group
 Prime Computer, Inc.
 Princeton University
 Programming Concepts, Inc.
 Purdue University
 Pyramid Technology
 Rabbit Software Corporation
 RAMTEC, Inc.
 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 Laboratories*
 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, Inc.
 Seybold Office Computing Group
 Shell Int. Pet. Mij.
 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 Magazine
 Software News
 Software Productivity Consortium
 Software Research Associates, Inc.
 Software-PEI
 SoHar
 Southwest Research Institute
 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
 Tennis Software Consulting, Inc.
 Texas Instruments, Inc.
 Texas Instruments-DSG
 Texas Internet Consulting
 The Algoma Steel Corporation Ltd.
 The C Journal, InfoPro Systems
 The Charles Stark Draper Laboratory, Inc.
 The Foxboro Company
 The Instruction Set
 The MITRE Corporation
 The Santa Cruz Operation
 The Wollongong Group
 Thinking Machines Corporation
 TIS
 Torch Computers Ltd.
 Toshiba Corporation
 Treasury Board of Canada*
 TRW
 Tsinghua University
 U.K. UNIX Systems User Group
 UNI Karlsruhe, Informatik II
 Uni'C' Computer Systems
 UniForum*
 UniForum Canada
 Unigram:X
 Unigroup of New York, Inc.
 Uni-Ops
 UNIQ Digital Technologies
 UniSoft Corporation
 UniSoft Ltd.
 Unisys Corporation**
 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
 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
 University of Wisconsin-Milwaukee
 University of Zagreb
 UNIX Houston
 UNIX International
 UNIX Review
 UNIX Systems
 UNIX Technologies
 UNIX User Group Austria
 UNIX Users of Minnesota
 UNIX World
 UNIX-C Club
 UNIX/WORLD
 US Air Force
 US Army
 US Army Ballistic Research Lab.
 US Army ISEC
 US Department of Defense
 US Department of H.U.D.
 US Dept. of Commerce NOAA/NOS
 US D.O.T. Systems Center
 US Federal Judicial Center
 US West Advanced Technologies
 USENIX Association*
 USSR State Committee for Standards
 Varian
 Venturcom
 Verdex 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
 XIOS Systems Corporation
 X/Open Company Ltd.
 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.

THIS PAGE WAS
BLANK IN THE ORIGINAL

The Source for UNIX Standardization Information

ISO/IEC 9945-1 : 1990 (POSIX .1) defines a standard operating system interface and environment based on the UNIX Operating System. It supports applications portability at the source-code level between multi-vendor computer systems.

POSIX.1 establishes a set of basic services fundamental to the efficient construction of applications programs. Access to these services is provided through an operating system using the C programming language. The OS interface establishes standard semantics and syntax, and allows for application developers to design portable applications.

Complementing existing standards related to computer languages, database management, and computer graphics, POSIX.1 is designed for and used by both application developers and system implementors. It provides a solid base for the systems procurement and evaluation processes. By specifying POSIX.1 confor-

mance, system purchasers can productively manage their software environments to achieve the benefits of applications portability. Likewise, hardware and software suppliers and developers will have a clear specification to follow in designing their systems and applications for the open software environment.

POSIX.1 constitutes a major step in the industry toward providing a comprehensive standardized applications environment. The IEEE's POSIX Committee is continuing POSIX-related standards work in areas such as POSIX-based Open System Environment (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), An Ada Language Binding for POSIX (IEEE Project 1003.5), and a FORTRAN Language Binding to POSIX (IEEE Project 1003.9).

UNIX is a registered trademark of AT&T.

IEEE Seminars . . . Keeping You Competitive Through Standards

If you define architectures for large distribution systems, set standards policy for your organization, or evaluate functional, performance, and integrity requirements, be sure to register for the new IEEE POSIX Seminar.

In Spring 1991, IEEE is launching a 2-day seminar based on the IEEE Project 1003.0, Guide to POSIX Open Systems Environments, and other IEEE projects concerning open systems. You will learn about frameworks and profiles for Information Technology standards, and how they directly apply to your organization.

For more information, call toll free in the USA at 1 (800) 678-IEEE and ask for Seminars on Standards. Or write to IEEE Standards Seminars, 445 Hoes Lane, PO Box 1331, Piscataway, NJ 08855-1331 USA.

ISBN 1-55937-061-0