INTERNATIONAL STANDARD

ISO/IEC 14515-1

IEEE Std 2003.1

First edition
2000-12-15

Information technology — Portable Operating System Interface (POSIX) — Test methods for measuring conformance to POSIX —

Part 1: System interfaces

Technologies de l'information — Interface de système de fonctionnement portable (POSIX) — Méthodes d'essai pour mesurer la conformité au POSIX —

Partie 1: Interfaces de système

Reference number
ISO/IEC 14515-1:2000(E)
IEEE
Std. 2003.1, 1992 edition
Reaffirmed 2000

ISO
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.ch
Web www.iso.ch

International Standard ISO/IEC 14515-1:2000(E)
IEEE Std 2003.1-1992 (Reaff 2000)

Information technology—Portable Operating System Interface (POSIX®)—Test methods for measuring conformance to POSIX—Part 1: System interfaces

Sponsor

Portable Applications Standards Committee
of the
IEEE Computer Society

Reaffirmed March 2000

IEEE-SA Standards Board

Approved November 2000

International Organization for Standardization
and by the
International Electrotechnical Commission

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

IEEE

American National Standard

Abstract: ISO/IEC 14515-1:2000(E) (IEEE Std 2003.1-1992) provides a definition of the requirements placed upon providers of POSIX test methods for POSIX.1 (ISO/IEC 9945-1:1990; IEEE Std 1003.1-1990). These requirements consist of a POSIX.1-ordered list of assertions defining those aspects of POSIX.1 that are to be tested and the associated test methods that are to be used in performing those tests. This standard is aimed primarily at POSIX.1 test suite providers and POSIX.1 implementors. This standard specifies those aspects of POSIX.1 that shall be verified by conformance test methods.

Keywords: assertion, assertion test, base assertion, conditional feature, extended assertion, POSIX. POSIX Conformance Document, POSIX Conformance Test Procedure, POSIX Conformance Test Suite, test method, test result code

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

Print: ISBN 0-7381-2691-8 SH94894
PDF: ISBN 0-7381-2692-6 SS94894

International Standard ISO/IEC 14515-1:2000(E)

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

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 3.

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

Attention is drawn to the possibility that some of the elements of this part of ISO/IEC 14515 may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

International Standard ISO/IEC 14515-1 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

ISO/IEC 14515 consists of the following parts, under the general title *Information technology — Portable Operating System Interface (POSIX) — Test methods for measuring conformance to POSIX*:

— *Part 1: System interfaces*

— *Part 2: Shell and utilities*

Annexes A and B form a normative part of this part of ISO/IEC 14515.

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

IEEE Std 2003.1-1992 (Reaff 2000)

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) 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 IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating 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-SA Standards Board
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
USA

Note: Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying patents for which a license may be required by an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

IEEE is the sole entity that may authorize the use of certification marks, trademarks, or other designations to indicate compliance with the materials set forth herein.

Introduction

(This introduction is not a normative part of IEEE Std 2003.1-1992, IEEE Standard for Information Technology—Test Methods for Measuring Conformance to POSIX—Part 1: System Interfaces, but is included for information only.)

This standard provides a definition of the requirements placed upon providers of a POSIX test method for the POSIX.1 {3} that are to be tested and the associated test methods that are to be used in performing those tests. This document is aimed primarily at POSIX.1 {3} test suite providers and POSIX.1 {3} implementors.this document specifies those aspects of the standard that shall be verified by conformance test methods.

Organization of This Standard

This document is organized into three portions:

1) Statement of scope, normative references, conformance requirements, and test methods (Section 1.)
2) Conventions and definitions (Section 2.)
3) Assertions to test POSIX.1 {3} interface facilities (Section 2. through 10.)

This introduction, any footnotes, notes accompanying the text, and the *informative* annexes are not considered part of this standard. Annexes A and B are informative.

Related Standards Activities

Activities to extend this stardard to address additional requirements are in progress, and similar efforts can be anticipated in the future.

The following areas are under active consideration at this time, or are expected to become active in the near future:[1]

1) Language-independent service descriptions of POSIX.1 {3}
2) C, Ada, and Fortran language bindings to (1)
3) Verification testing methods
4) Realtime facilities
5) Secure/Trusted System considerations
6) Network interface facilities
7) System Administration
8) Graphical User Interfaces
9) Profiles describing application- or user-specific combinations of Open Systems standards for: supercomputing, multiprocessor, and batch extension; transaction processing; realtime systems; and multiuser systems based on historical models
10) An overall guide to POSIX-based or related Open Systems standards and profiles

Extensions are approved as “amendments” or “revisions” to this document, following IEEE and ISO/IEC procedures.

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

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

[1]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.

IEEE Std 2003.1-1992 was prepared by the P2003.1 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 P2003.1 working group was follows:

Technical Commitee on Operating Systems and Application Environments (TCOS)

Chair:	Jehan-François Pâris

TCOS Standards Subcommittee

Chair:	Jim Isaak
Vice-Chairs:	Ralph Barker
	Hal Jespersen
	Lorraine C. Kevra
	Pete Meier
	Andrew T. Twigger
Treasurer:	Quin Hahn
Secretary:	Shane P. McCarron

P2003.1 Working Group Officials

Chair:	Roger J. Martin
Vice-Chairs:	Nicholas Ray Wilkes (1989–1990)
	Carol Raye (1987–1988)
Editor:	Anthony V. Cincotta
Secretaries:	Lowell G. Johnson (1989–1990)
	Doris Lebovits (1987–1988)

P2003.1 Technical Reviewers

Sanjay Agraharam	James M. Moe	Andrew T. Twigger
Anthony V. Cincotta	Anita Mundkur	Bruce Weiner
Steve Henderson	Gerald Powell	
Mark Lamonds	Ravi Tavakley	

P2003.1 Working Group

Sanjay Agraharam	Lorraine C. Kevra	Fred Noz
James P. Bound	Martin J. Kirk	Bob Palm
Kenneth R. Gibb	Mark Lamonds	Gerald Powell
Judy Guist	Robert M. Lenk	Ravi Tavakley
Rich Hamm	Kevin Lewis	Donn S. Terry
Steve Henderson	Martin J. McGowan III	Andrew T. Twigger
Scott Jameson	James M. Moe	Bruce Weiner
Greg Jones	Anita Mundkur	Brian Weis

The following persons provided valuable input during the balloting period:

Chuck Karish	Paul Rabin

The following persons were members of the P2003.1 balloting that approved this standard for submission to the IEEE Standards Board:

Mike Lambert	*X/Open Organizational Representative*
Shane P. McCarron	*Unix International Organizational Representative*
Fritz Schulz	*OSF Organizational Representative*

David Athersych
Geoff Baldwin
Jerome E. Banasik
Robert Barned
Kabekode V.S. Bhat
Robert Bismuth
James Bohem
Robert Borochoff
Keith Bostic
James P. Bound
Phyllis Eve Bregman
A. Winsor Brown
Fred Lee Brown, Jr.
Nicholas A. Camillone
Clyde Camp
John Caywood
Kilnam Chon
Chan F. Chong
Anthony V. Cincotta
Robert L. Claeson
Richard Cornelius
William M. Corwin
Mike Cossey
William Cox
Donald W. Cragun
Ava Maria De Alvare
Terence Dowling
Stephen A. Dum
John D. Earls
Ron Elliott
Philip W. Enslow
Kenneth T. Faubel
Terence Fong
Kenneth R. Gibb
Michel Gien
Gregory W. Goddard
Dave Grindeland

Judy Guist
Charles E. Hammons
Allen Hankinson
Craig Harmer
Dale Harris
John L. Hill
David F. Hinnant
Lee A. Hollaar
Terrence W. Holm
Richard Hughes-Rowlands
Jim Isaak
Dan Iuster
Scott Jameson
Hal Jespersen
Lowell G. Johnson
Greg Jones
Lorraine C. Kevra
Jeffrey S. Kimmel
Martin J. Kirk
Dale L. Kirkland
Kenneth C. Klingman
D. Richard Kuhn
Robin B. Lake
Mark Lamonds
Doris Lebovits
Robert M. Lenk
David Lennert
Kevin Lewis
Joseph F. P. Luhukay
Robert J. Makowski
Roger J. Martin
Joberto S. B. Martins
Martin J. McGowan III
Robert W. McWhirter
Paul Merry
Doug Michels
James M. Moe

Anita Mundkur
Martha Nalebuff
Barry Needham
Fred Noz
John C. Penney
P. J. Plauger
Gerald Powell
Scott E. Preece
James M. Purtilo
Carol Raye
Christopher J. Riddick
Robert Sarr
Norman Schneidewind
Leonard W. Seagren
Karen Sheaffer
Dan Shia
Mukesh Singhal
Richard Sniderman
Douglas H. Steves
Scott A. Sutter
Robert Switzer
Ravi Tavakley
Donn S. Terry
Gary F. Tom
Andrew T. Twigger
Mark-Rene Uchida
Michael W. Vannier
John W. Walz
Alan G. Weaver
Bruce Weiner
Brian Weis
Peter J. Weyman
Andrew E. Wheeler, Jr.
Nicholas Ray Wilkes
Oren Yuen
Alaa Zeineldine
Jason Zions

When the IEEE Standards Board approved this standard on December 3, 1992, it had the following membership:

Marco W. Migliaro, *Chair*
Donald C. Loughry, *Vice Chair*
Andrew G. Salem, *Secretary*

Dennis Bodson	Donald N. Heirman	T. Don Michael*
Paul L. Borrill	Ben C. Johnson	John L. Rankine
Clyde Camp	Walter J. Karplus	Wallace S. Read
Donald C. Fleckenstein	Ivor N. Knight	Ronald H. Reimer
Jay Forster*	Joseph Koepfinger*	Gary S. Robinson
David F. Franklin	Irving Kolodny	Martin V. Schneider
Ramiro Garcia	D. N. “Jim” Logothetis	Terrance R. Whittemore
Thomas L. Hannan	Lawrence V. McCall	Donald W. Zipse

*Member Emeritus

Also included are the following nonvoting IEEE Standards Board liaisons:

Satish K. Aggarwal	Richard B. Engelman	Stanley Warshaw
James Beall	David E. Soffrin	

Mary Lynne Nielsen
IEEE Standards Project Editor

With the recommendation and endorsement of the standard developers, this standard is dedicated to the Nicholas Ray Wilkes, who served as vice-chair of IEEE Std 1003.3-1991.

CLAUSE PAGE

CLAUSE PAGE

INTERNATIONAL STANDARD ISO/IEC 14515-1:2000(E)

Information technology—Portable Operating System Interface (POSIX®)—Test methods for measuring conformance to POSIX—Part 1: System interfaces

1. General

1.1 Scope

This standard defines the general requirements and test methods for measuring conformance to ISO/IEC 9945-1 :1990 (IEEE Std 1003.1-1990) hereinafter referred to as "POSIX.1" {3}.[1] It also defines the test assertions for measuring conformance to POSIX.1 {3}.

This standard is intended for use by

- Developers of POSIX.1 {3} test methods;
- Implementors of POSIX.1 {3} implementations;
- Application writers for POSIX.1 {3} conforming implementations;
- POSIX.1 {3} testing laboratories; and
- Others interested in validating the conformance of a vendor-claimed POSIX.1 {3} implementation

The purpose of this standard is to specify the test assertions and related test methods for measuring conformance of an implementation to POSIX.1 {3}.

Testing conformance of an implementation to a standard includes testing the claimed capabilities and behavior of the implementation with respect to the conformance requirements of the standard. These test methods are intended to provide a reasonable, practical assurance that the implementation conforms to the standard. Use of these test methods will not guarantee conformance of an implementation to POSIX.1 {3}; that normally would require exhaustive testing, which is impractical for both technical and economic reasons.

The technical specifications for a POSIX System Application Program Interface are defined in POSIX.1 {3}. IEEE Std 2003.1-1992 defines a means of measuring conformance to the POSIX.1 {3} technical specifications. Any question of

[1]The numbers in curly braces correspond to those of the references in 1.2.

interpretation of those technical specifications arising from the use of this standard is a question of interpretation of POSIX.1 {3}.

1.2 Normative References

The following standards contain provisions that, through references in this text, constitute provisions of this standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this standard are encouraged to investigate the possibility of applying the most recent editions of the standards listed below.

{1}ISO/IEC 646: 1991 [2] *Information technology—ISO 7-bit coded character set for information interchange.*

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

{3}ISO/IEC 9945-1:1990 (IEEE Std 1003.1-1990) [3] *Standard for Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language].*

{4}IEEE Std 1003.3-1991 *IEEE Standard for Information Technology—Test Methods for Measuring Conformance to POSIX.*

1.3 Conformance

1.3.1 Implementation Conformance

1.3.1.1 Requirements

NOTE — The types of assertion numbering used in this document are explained in IEEE Std 1003.3-1991 {4}.

D01(A) The conformance document defines an environment in which an application can be run with the behavior specified by POSIX.1 {3} in 1.3.1.1 of the PCD.1[4].

1.3.1.2 Documentation

D02(A) The conformance document contains a statement that indicates the full name, number, and date of the POSIX.1 {3} standard that applies in 1.3.1.2 of the PCD.1.

D03(C) If the implementation makes international software standards available to a conforming application and the implementation documents this:

The list of these standards is contained in 1.3.1.2 of the PCD.1.

NOTE — The conformance document requirements are specified in 1.4.17.1.

1.3.2 Application Conformance

There are no test methods provided for this subclause.

1.3.3 Language-Dependent Services for the C Programming Language

D04(C) If the C-language provided does not conform to the C Standard {2}:

[2]ISO/IEC documents can be obtained from the ISO office, 1, rue de Varembé, Case Postale 56, CH-1211, Genève 20, Switzerland/Suisse.
[3]IEEE documents can be obtained from the The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, New York 10017, USA.
[4]Abbreviations are defined in 1.4.2.3.

The conformance documentation states that the C-language provided does not conform to the C Standard {2} in 1.3.3 of the PCD.1.

1.3.3.1 Types of Conformance

There are no test methods provided for this subclause.

1.3.3.2 C Standard Language-Dependent System Support

D05(C) If the C Standard {2} language binding is supported:

The conformance document states the version of the C Standard {2} referenced in fulfilling the requirements of Section 8. of POSIX.1 {3} in 1.3.3.2 of the PCD.1.

1.3.3.3 Common-Usage C Language-Dependent System Support

DGA01 For all functions stated in Section 8.1 of POSIX.1 {3}:

The details of all differences between the interface provided and the interface that would have been provided had the C Standard {2} been implemented instead of Common-Usage C are contained in 8.1 of the PCD.1.

D06(C) If the Common-Usage C language binding is supported and differences exist from the C Standard {2}:

The version of the C Standard {2} used for referencing the differences is stated in 1.3.3.3 of the PCD.1.

1.3.4 Other C Language-Related Specifications

NOTE — When a macro exists for any function, except *assert*(), *setjmp*(), and *sigsetjmp*(), then all assertions, except those in the Synopsis clause, are tested for both the macro and the function interfaces. The identifiers with external linkage providing access to *setjmp*() and *sigsetjmp*() shall be tested. Assertions in the Synopsis clause are specifically related to the function or macro interfaces as appropriate. (See 1.4.8.)

GA01 For all elements except *abort*(), *assert*(), *getc*(), *putc*(), *setjmp*(), *sigsjmp*(), and *tzset*():

If the function is defined as a macro:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary.

For *funct*() of *setjmp*() and *sigsetjmp*():

If *funct*() is defined as a macro:

It protects its result value with extra parentheses when necessary.

There are no assertions specific to this subclause.

1.3.5 Other Language-Related Specifications

There are no test methods provided for this subclause.

1.3.6 Test Methods Conformance

1.3.6.1 Requirements

A conforming implementation of POSIX.3.1 test methods shall meet all of the following criteria:

1) All test method requirements of 1.4.4 through 1.4.19 that are associated with required base assertions shall be implemented.

2) If the test method tests for a specific POSIX.1 {3} profile, then all test method requirements of 1.4.4 through 1.4.19 that are associated with the requirements of this profile shall be implemented.
3) All required base assertions shall be tested.
4) If the PCTS.1 tests for a specific POSIX.1 {3} profile, then all conditional base assertions associated with the requirements of this profile shall be implemented.
5) A conforming PCTS.1 shall conform to the requirements of IEEE Std 1003.3-1991 (POSIX.3 {4}).
6) A conforming PCTS.1 shall document the limitations on POSIX.1 {3} testing when employing the PCTS.1. *Example:* This PCTS.1 considers appropriate privilege to be implemented monolithically as a single privilege.

1.3.6.2 Recommendations

A conforming implementation of POSIX.3.1 test methods should meet the following criterion:

— If extended assertions are within the POSIX.1 {3} profile being tested, then these assertions should, when possible, be tested.

1.4 Test Methods

1.4.1 Introduction

This clause defines the test methods to be used to measure conformance to POSIX.1 {3} and contains requirements and recommendations for the test methods specific to testing POSIX.1 {3}.

The contents of POSIX.3.1 are C-language specific because POSIX.1 {3} is specified in terms of C-language bindings. Except where explicitly stated, the requirements and recommendations of POSIX. 3.1 apply to a PCTS.1 written in either Common-Usage C or C Standard {2} and test conformance to either Common-Usage C or C Standard {2} language bindings of a target system.

The test methods used for POSIX.1 {3} conformance measurement shall consist of a PCTS.1 and a documentation audit of the POSIX Conformance Document as required by POSIX.1 {3} and its associated PCTP. 1.

1.4.2 Definitions

1.4.2.1 Terminology

1.4.2.1.1 may: An option for *test methods.*

In this standard the negative of *may* is *need not*.

1.4.2.1.2 shall: A requirement for *test methods.*

1.4.2.1.3 should: A recommended practice for *test methods*.

1.4.2.2 General Terms

1.4.2.2.1 assertion: A statement of functionality or behavior for a POSIX *element* that is derived from the POSIX standard being tested and that is true for a conforming POSIX implementation. [IEEE Std 1003.3-1991]

1.4.2.2.2 assertion number: The numeric identifier assigned to an *assertion*.

The name of the *element* and the *assertion number* together uniquely identify an *assertion*. [IEEE Std 1003.3-1991]

1.4.2.2.3 assertion test: The software or procedural methods that ascertain the conformance of a POSIX implementation to an *assertion*. [IEEE Std 1003.3-1991]

1.4.2.2.4 base assertion: An *assertion* that is required to be tested for required features and for implemented *conditional features*. [IEEE Std 1003.3-1991]

1.4.2.2.5 conditional feature: A feature or behavior referred to in a POSIX standard that need not be present on all conforming implementations. [IEEE Std 1003.3-1991]

1.4.2.2.6 development system: The computer system used to compile and configure a PCTS.1. [IEEE Std 1003.3-1991]

1.4.2.2.7 element: A functional interface or a namespace allocation.

Examples of elements are C functions or utility programs. Examples of namespace allocation include headers or error return value constants. [IEEE Std 1003.3-1991]

1.4.2.2.8 extended assertion: An *assertion* that is not required to be tested. [IEEE Std 1003.3-1991]

1.4.2.2.9 modem control: The monitoring of modem status lines.

1.4.2.2.10 POSIX Conformance Document (PCD): The conformance document required by a POSIX standard. [IEEE Std 1003.3-1991]

1.4.2.2.11 POSIX Conformance Test Procedure (PCTP): The nonsoftware procedures possibly used in conjunction with other test methods to measure conformance. [IEEE Std 1003.3-1991]

1.4.2.2.12 POSIX Conformance Test Suite (PCTS): The collection of software possibly used in conjunction with other test methods to measure conformance. [IEEE Std 1003.3-1991]

1.4.2.2.13 required feature: Either a single facility or behavior, or one of a pair of alternative facilities or behaviors, required by a POSIX standard that is always present on a conforming implementation. [IEEE Std 1003.3-1991]

1.4.2.2.14 target system: The combination of the computer system on which the PCTS is executed and the parts of the *development system* that are used to generate the executable code of a PCTS. [IEEE Std 1003.3-1991]

1.4.2.2.15 test method: The software, procedures, or other means specified by a POSIX standard to measure conformance.

Test methods may include a PCTS, PCTP, or an audit of a PCD. [IEEE Std 1003.3-1991]

1.4.2.2.16 test result code: A value that describes the result of an *assertion test*. [IEEE Std 1003.3-1991]

1.4.2.2.17 unattainable limit: A limit that is undefined for a *target system* or that has a magnitude exceeding the POSIX.1 {3} specified minimum and that would require an unreasonable amount of time or system resources to test.

1.4.2.3 Abbreviations

For the purposes of this standard, the following abbreviations apply:

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

1.4.2.3.2 Common-Usage C: Common-Usage C Language-Dependent System Support.

1.4.2.3.3 PCD.1: POSIX Conformance Document for POSIX.1 {3}.

1.4.2.3.4 PCTP. 1: POSIX Conformance Test Procedures for POSIX.1 {3}.

1.4.2.3.5 PCTS.1: POSIX Conformance Test Suite for POSIX.1 {3}.

1.4.2.3.6 POSIX: Colloquial name for the collection of IEEE 1003 standards, draft standards, and projects.

1.4.2.3.7 POSIX.1: ISO/IEC 9945-1 :1990 (IEEE Std 1003.1-1990).

1.4.2.3.8 POSIX.3: IEEE Std 1003.3-1991.

1.4.2.3.9 POSIX.3.1: The POSIX.3 standard that defines the test methods for the POSIX.1 {3} standard; a reference to this standard.

1.4.3 Assertion Concepts

1.4.3.1 Security Features

The assertions in this standard, in general, do not address the conditional features associated with extended security controls, alternate file access control mechanisms, and additional file access control mechanisms.

1.4.3.2 Appropriate Privileges

All assertions that require either the presence or absence of appropriate privilege state their requirements. Assertions that contain no stated requirement relating to appropriate privilege hold true regardless of what appropriate privileges the calling process has.

1.4.3.3 Testing Constraints

A testing constraint is a conditional feature, behavior, or additional hardware or software needed to test an assertion. Testing constraints are employed within the classification of the assertion with the syntax:

(testing constraint?classification:UNTESTED)

1.4.3.4 PCD Announcement Mechanism

When assertions are dependent on the PCD.1 for an announcement mechanism about the support or nonsupport of POSIX.1 {3} conditional features, then the conditional phrase of the assertion shall incorporate the appropriate PCD symbol (see Table 1.4).

1.4.4 PCTS Execution

1.4.4.1 PCTS Target System Support Facilities

Support facilities are required to establish an operational environment for some tests within a PCTS.1, where it is necessary to undertake tasks that are beyond the scope of POSIX.1 {3}. The implementation of these support facilities will differ between different target systems, and it will be necessary for the user of the PCTS.1 to provide an interface to the support facilities to be able to execute the PCTS.1 fully.

1.4.4.1.1 Requirements

The interface syntax for a target system support facility shall be specified in the PCTS.1 documentation. The values designating successful and unsuccessful provision of the requested facility shall also be specified in the PCTS.1 documentation. A target system support facility shall return an unsuccessful indicator if the target system does not support the facility.

1.4.4.1.2 Recommendations

Where POSIX.1 {3} does not provide the means for establishing the required operational environment, a portable PCTS.1 should not make assumptions about the functions a target system will use to establish this environment. To establish the test environment, the PCTS.1 should use well-defined target system support facilities that may have to be provided by the user of the PCTS.1.

To increase portability, a PCTS.1 should allow the test installer to provide target system support facilities to establish an environment for testing.

A target system support facility should be used to control

— Associating a controlling terminal with a process
— Establishing appropriate privileges for a process
— Mounting a file system with read and write capabilities
— Mounting a file system with read only capability
— Unmounting a file system
— Setting S_ISUID mode with *chmod*()
— Setting S_ISGID mode with *chmod*()

The PCTS.1 should not require information about the manner in which the implementation provides the requested facilities nor about the time taken to provide these facilities.

1.4.4.2 Test Tolerances

1.4.4.2.1 Requirements

In order to test for proper functioning of time-related functions, a PCTS.1 shall permit allowances for a degree of variability in the time to perform certain operations. For example, comparison of the times returned by the *times*() function to the times returned separately by the child processes will vary slightly between separate executions of the test.

POSIX.1 {3} does not define the timeliness or the asynchronous nature of certain events. The PCTS.1 shall expect that the effect of an event on both the initiating and all target processes shall be completed in a reasonable amount of time and that all of the affected processes shall continue in the expected manner after the event has occurred. The PCTS.1 shall report or document the time variation that it allows for a test tolerance.

1.4.4.2.2 Recommendations

None.

1.4.4.3 Timeout Tolerances

1.4.4.3.1 Requirements

In order to test certain time-related events, a PCTS.1 shall permit allowances for variance in the time at which the timer goes off. For instance, the *alarm*() function sets a timer that will cause a SIGALRM to be sent at the specified time. Variations in system load and process scheduling can cause the actual time at which SIGALRM is sent to vary.

1.4.4.3.2 Recommendations

A PCTS.1 should use a user configuration variable to specify the defined amount of time beyond the specified timer time that a test should wait before reporting failure.

1.4.5 PCTS Testing Constraints

1.4.5.1 Requirements

The symbols to be used by a PCTS.1 when denoting POSIX.3.1 testing constraints are specified in Table 1.1.

Table 1.1—Testing Constraints

Symbol	Testing Constraint
PCTS_APP_LINK_DIR	The implementation permits the appropriate privilege process to create links to directories.
PCTS_GTI_DEVICE	The implementation provides device types that support the general terminal interface.
PCTS_GTI_BUFFERS_OUTPUT	The general terminal interface device buffers its output.
PCTS_PROCESS_LIMIT	The system process limit is not exceeded before {CHILD_MAX} is reached.
PCTS_CHMOD_SET_IDS	The function *chmod*() allows setting the S_ISUID and S_ISGID modes.
PCTS_INVALID_SIGNAL	An invalid or unsupported signal number exists.
PCTS_INVALID_OWNER	An invalid group or owner ID exists.
PCTS_APP_MODE	The implementation permits the appropriate privilege process to change file mode.
PCTS_APP_OWNER	The implementation permits the appropriate privilege process to change owner and group IDs.
PCTS_APP_TIMES	The implementation permits the appropriate privilege process to change file times.
PCTS_ROOT_WRITABLE	The root file system is not read-only.
PCTS_HPA_FILE	The implementation supports the high-performance attribute file.

1.4.5.2 Recommendations

None.

1.4.6 PCTS Limits and Configuration Variables

1.4.6.1 Requirements

Table 1.2 defines sets of limits. The POSIX.1{3} values {ARG_MAX}, {CHILD_-MAX}, {LINK_MAX}, {MAX_CANON}, {MAX_INPUT}, {NAME_MAX}, {OPEN_MAX}, {PATH_MAX}, {PIPE_BUF}, and {TZNAME_MAX} used by the PCTS.1 shall be obtained from the *pathconf*() and *sysconf*() calls, unless the assertion explicitly states that the value is to be obtained from a specific header. The POSIX.1 {3} limit value for {PCTS_LOCK_ MAX} is imposed by the system.

Because certain of these POSIX.1 {3} limits, for practical testing purposes, may be unattainable or require an unreasonable amount of time or resources before they are attained, values for the mimimurn PCTS.1 testing limit (PCTS_mtl) are provided as specified in Table 1.2. When a specific PCTS_mtl is less than the associated POSIX.1 {3} limit, a PCTS.1 may set the actual test limit value (PCTS_tlv) such that PCTS_mtl is less than or equal to PCTS_tlv and PCTS_tlv is less than or equal to the POSIX.1 {3} limit. If the implementation does not provide a POSIX.1 {3} limit, the PCTS.1 shall set the test limit value such that PCTS_mtl is less than or equal to PCTS_tlv.

When the POSIX.1 {3} limit obtained from *pathconf*(), *syscon*(), or another system-imposed limit is less than or equal to the associated PCTS_mtl, the POSIX.1 {3} limit shall be used for testing, as shown in Table 1.2.

Table 1.2—Configuration Variables

PCTS Limit Name	POSIX.1 Limit	PCTS_mtl (PCTS.1 minimum test limit)
PCTS_ARG_MAX	{ARG_MAX}	The lesser of {ARG_ MAX}, as obtained from *sysconf*(), and ten times {_POSIX_ARG_MAX}.
PCTS_CHILD_MAX	{CHILD_MAX}	The lesser of {CHILD_MAX}, as obtained from *sysconf*(), and 256.
PCTS_LINK_MAX	{LINK_MAX}	The lesser of {LINK_MAX}, as obtained from *pathconf*(), and 256.
PCTS_LOCK_MAX	system-imposed	The lesser of the system-imposed limit and 2500.
PCTS_MAX_CANON	{MAX_CANON}	The lesser of {MAX_CANON}, as obtained from *pathconf*(), and four times {_POSIX_MAX_CANON}.
PCTS_MAX_INPUT	{MAX_INPUT}	The lesser of {MAX_INPUT}, as obtained from *pathconf*(), and four times {_POSIX_MAX_INPUT}.
PCTS_NAME_MAX	{NAME_ MAX}	The lesser of {NAME_MAX}, as obtained from *pathconf*(), and 2048.
PCTS_OPEN_MAX	{OPEN_MAX}	The lesser of {OPEN_MAX}, as obtained from *sysconf*(), and 256.
PCTS_PATH_MAX	{PATH_MAX}	The lesser of {PATH_MAX}, as obtained from *pathconf*(), and 4096.
PCTS_PIPE_BUF	{PIPE_BUF}	The lesser of {PIPE_BUF}, as obtained from *pathconf*(), and 32767.
PCTS_TZNAME_MAX	{TZNAME_MAX}	The lesser of {TZNAME_MAX}, as obtained from *sysconf*(), and 256.

The term *obtained from* in Table 1.2 refers to the meaning of the return value from a call to *pathconf*() or *sysconf*(). If this return value is −1 and the error indicator is unchanged, the PCTS.1 shall interpret this as a value greater than the corresponding PCTS_tlv.

The manner in which the assertion is modified is described in each of the assertions affected.

A PCTS.1 shall use configuration parameters to provide information about the control sequences used to perform specific terminal operations, as shown in Table 1.3. The sequences that an implementation uses to erase characters and lines differ from implementation to implementation. A PCTS.1 can assume no knowledge of these character sequences. The PCTS.1 will verify that the character sequence used to erase the character/line matches the expected sequence for the character/line erase operation for the target system. These character sequences should be provided to the PCTS.1 using the configuration variables specified in Table 1.3.

Table 1.3—Configuration Control Sequences

Test Option	Value
PCTS_ECHOE	Erase character sequence, or empty if none.
PCTS_ECHOK	Erase line sequence, or empty if none.

The testing of character and line erase sequences is undertaken in canonical mode. The PCTS.1 shall set up the test for character erasure by first ensuring that the terminal input line is empty and then placing a sequence of characters into the input line. The PCTS.1 shall then issue an erase character command and ensure that the sequence specified in {PCTS_ECHOE} is used to erase the character. A similar technique is used to test for line erasure except the sequence specified in {PCTS_ ECHOK} is used to check that the correct sequence of characters is used to erase the line.

1.4.6.1.1 Recommendations

None.

1.4.7 Test Methods for Headers

1.4.7.1 Requirements

The symbol _POSIX_SOURCE shall be defined by the PCTS.1 prior to the inclusion of any headers.

Since headers are not required to be readable files of C source code on the target system, it is necessary to use indirect ways to test their content. The items defined in headers shall be tested as follows to establish compliance as required in POSIX.1 {3}.

- *Symbolic constants*, *macros*, and *types* shall be tested for existence and their functional correctness. See 1.4.8 for more requirements on macros.
- *Function prototypes* in the C Standard {2} shall be tested for their existence and correctness of definition.
- *Function declarations* in Common-Usage C shall be tested for their existence and correctness of definition.
- *Function return values* in Common-Usage C for functions not returning integer type shall be tested for correctness of return value.
- *Data aggregate definitions* shall be tested for their existence and correctness of all members and types associated with members that are specified in POSIX.1 {3}.
- *Variables* shall be tested for their existence and types.
- The ability to include a header more than once shall be tested.
- The ability to access a symbol whose definition is required in more than one header shall be tested.

1.4.7.2 Recommendations

Headers should be tested by including them in test programs. The following criteria should be used for type checking.

- For a definition of a structure type, the size of each structure member should be compared to the size of an item of the type of the member or to the size of the type of the member itself.
- For a variable declaration, the size of the variable should be compared to the size of a variable of the specified type or to the size of the specified type itself. This test applies to members of structures, not to structures themselves.
- For a variable, the variable should be compared to another variable of the same type. This test applies to members of a structure, not to structure variables.
- For a variable, the variable should be assigned to another variable of the same type.
- For a variable, the address of the variable should be assigned to another variable of the type pointer to the defined type. This does not apply to function return values.

The detection of a nonconformity relies upon the **C** compiler generating an error or warning message. If such a message is generated, then there should be either a PCTP.1 to check the compiler output to resolve whether the tests **PASS** or **FAIL**, or software to do the checking automatically.

To test for proper behavior of duplicate header inclusion and duplicate symbolic definition in headers, a test should include all of the headers to be tested in a valid order such that there are duplicate inclusions of all headers.

If the PCTS.1 uses a language compiler or translator to accomplish testing of the assertions related to headers, the PCTS.1 should not rely solely on the return value from that compiler or translator, but should also examine any messages produced during test execution.

1.4.8 Test Methods for Macros

1.4.8.1 Requirements

A PCTS.1 shall test that any interface that is implemented as a macro has an underlying function of the same name (if required by the standard) and that both the macro and the function behave in the same manner with respect to the assertions.

The PCTS.1 shall also test that the macro is implemented in such a manner that it evaluates its arguments only once, fully protected by parentheses when necessary, and yields a resulting value that can be used in complex expressions without requiring extra parentheses.

1.4.8.2 Recommendations

None.

1.4.9 Test Methods for Invalid Constant Values

1.4.9.1 Requirements

Some POSIX.1 {3} functions take a symbolic constant as an argument and have a specified behavior when the argument has an invalid value. Because an implementation can have extensions that permit the function to accept values other than those specified by POSIX.1 {3}, a portable PCTS.1 shall not simply choose such a value.

1.4.9.2 Recommendations

A PCTS.1 should use a user configuration variable to select an invalid value of each constant for which the behavior of invalid values is tested.

1.4.10 Test Methods for *errno* Checking

1.4.10.1 Requirements

A test shall be performed for detection of all values of *errno* specified in the Errors subclause of each element. If more than one error occurs in processing a function call, an assertion test shall not depend on the order in which the errors are detected or in which error code is returned.

Tests for error code detection shall be coded to generate only a single error code when possible.

1.4.10.2 Recommendations

None.

1.4.11 Test Methods for File Accessibility

1.4.11.1 Requirements

POSIX.1 {3} defines the cases in which access is granted to a file from a POSIX.1 {3} interface both in the case that the implementation provides a mechanism for a process to exist with the appropriate privilege to override the standard file access control mechanism and the case that the process has standard file access capabilities. A PCTS.1 shall test for the accessibility of a file in both of these cases and shall test for the [EACCES] error indication in the case that standard file access is denied to a process. These tests shall be carried out for each of the interfaces that contain assertions refering to the ability to access a file or assertions refering to the [EACCES] error.

POSIX.1 {3} defines the cases of additional and alternate file access control mechanisms without defining the manner in which these mechanisms can be activated. A PCTS.1 is not required to test these mechanisms, but shall run in an environment where these mechanisms exist but are not enabled.

1.4.11.1.1 Recommendations

None.

1.4.12 Test Methods for Signal Functions

1.4.12.1 Reentrance of Interrupted Functions

1.4.12.1.1 Requirements

A PCTS.1 shall test that, when a function in a quiescent state is interrupted by a signal, a call to the interrupted function from the signal handling routine behaves correctly and does not improperly affect the result of the interrupted function.

1.4.12.1.2 Recommendations

None.

1.4.12.2 Functions Interruptable by Signals

1.4.12.2.1 Requirements

When executing a test to verify that *errno* is set to [EINTR] when a function is interrupted, a PCTS.1 shall not use the default action of job control signals to effect the interruption. These signals cause the process to stop and do not actually interrupt the function.

1.4.12.2.2 Recommendations

None.

1.4.12.3 *sigaction*() Testing

1.4.12.3.1 Requirements

See *sigaction*() assertions.

1.4.12.3.2 Recommendations

If the behavior associated with {_POSIX_JOB_CONTROL} is supported, the recommended test method is to establish a known value for a globally accessible variable and then to call *sigaction*() for the SIGSTOP signal to attempt to associate a signal catching function with this signal. The handler would be written to change the known value. A SIGSTOP signal should then be generated, followed by a SIGCONT signal to cause the process to continue. If the known value has changed, the handler was established even though *sigaction*() failed.

If the behavior associated with {_POSIX_JOB_CONTROL} is not supported, the recommended method for testing that the signal catching function is not associated with the signal when *sigaction*() fails is to install a signal-catching function for SIGKILL that changes the value of a global variable. Then a SIGKILL signal is sent to the process. If the process can check whether the global variable has been changed (even determining that it has not), SIGKILL has not been delivered properly.

Possible tests include

— Having the handler do an *_exit*(*n*) and having the parent check exit status
— Writing to the file system from the handler

1.4.13 Test Methods for Files and Input/Output Primitives

1.4.13.1 Requirements

A PCTS.1 shall use at least one of the file types that are defined in POSIX.1 {3} when testing an assertion that applies to files. The file type chosen may differ for different assertions. Wherever a specific set of file types are defined in an assertion, a PCTS.1 shall use each of the file types stated.

1.4.13.2 Recommendations

A PCTS.1 implementor may choose to improve test coverage by increasing the number of file types covered by assertions.

1.4.14 File Descriptor Closure Test Method

1.4.14.1 Requirements

A PCTS.1 shall test that file descriptors are closed by *_exit*(), *exit*(), and *abort*(). (See GA41 in 3.2.)

1.4.14.2 Recommendations

The recommended method for testing file descriptor closure on a regular file is as follows:

— Parent process opens a file, then creates a child with *fork*().
— The child process creates an exclusive lock on the file and calls *_exit*().
— Upon termination of its child process, the parent attempts to exclusively lock the same part of the file. If it can, the test passes. If it cannot, the test fails.

The recommended method for testing this on a pipe (or FIFO) is as follows:

1) Create a pipe.
2) Fork a child process.
3) Close the write channel of the pipe in the parent process.
4) Terminate the child process.
5) Expect a *read*() on the pipe in the parent process to return a value of 0.

1.4.15 Test Methods for Terminal I/O Functions

1.4.15.1 Requirements

Terminal I/O testing shall not be performed using an interactive test method, and the implementation shall provide a method whereby a process can acquire a controlling terminal.

1.4.15.2 Recommendations

Terminal I/O should be tested using an automated test method. This method provides for accuracy and repeatability of test results by eliminating the need for human intervention while running a PCTS.1. An automated method of terminal I/O testing requires two asynchronous ports to be electrically connected in closed-loop mode and for these ports to support modem control protocols. One of the closed-loop ports is the driver port and the other is the target port. The terminal I/O tests send character data between the driver port and the target port via the connecting cable as though the driver port were a terminal. The tests then compare the data received with the expected results.

Target systems normally configured with less than two asynchronous ports may be specifically configured with two asynchronous ports for the terminal I/O tests. The normal configuration will be considered to have passed the terminal I/O tests if the special two-asynchronous-port configuration has passed the tests.

A more limited method that may be used if only one asynchronous port is available is to wire the port to loop the output of the port back to the input of the same port.

A PCTS.1 may determine the attributes of an asynchronous port through the use of the *tcsetattr*() and *tcgetattr*() functions. An attribute, such as CLOCAL for modem control, may be set using *tcsetattr*(). If that value is not set after being set by *tcsetattr*() and read by *tcgetattr*(), then modem control is not supported by that serial port.

1.4.16 Test Methods for File Formats

1.4.16.1 Requirements

The PCTS.1 shall supply all files used to test file formats (see Section 10.). These file formats are tested using the file reading utility. Do not create them on the target system with the file creating utility.

Where the PCTS.1 tests the content of text in file format headers, it shall do so using the ISO/IEC 646 :1991 {4} IRV character representation (see Section 10.).

1.4.16.2 Recommendations

None.

1.4.17 Documentation Audit

1.4.17.1 Requirements

The purpose of the documentation audit is to verify that the PCD.1 meets the requirements specified in Section 1.3.1.2 of POSIX.1 {3} as enumerated by the documentation assertions. These general requirements are that the PCD.1

- Has the same structure as POSIX.1 {3}, with the information presented in the appropriately numbered sections, clauses, and subclauses. Information may appear under a higher level heading or in a section, clause, or subclause whose content in POSIX.1 {3} is intended to cover multiple sections, clauses, or subclauses throughout the standard.
- Does not contain information about extended facilities or capabilities outside the scope of POSIX.1 {3} and only contains information related to the statements in Section 1.3.1.2 of POSIX.1 {3}
- Describes the limit values found in the `<limits.h>` and `<unistd.h>` headers and listed in Sections 2.8 and 2.9 of POSIX.1 {3}, stating values, the conditions under which those values may change, and the limits of such variations, if any
- Describes the behavior of the implementation for all implementation-defined features defined in POSIX.1 {3}
- May specify the behavior of the implementation for those features where POSIX.1 {3} states that implementations may vary or implies variations among implementations
- May specify the behavior of the implementation for those features where features are identified as undefined or unspecified, or where the POSIX.1 {3} text implies these terms
- Specifies all items of POSIX.1 {3} where the phrases “shall document” or “shall be documented” are used

Extended security controls may necessitate features to be documented that do not have explicit documentation assertions. Such additional documentation shall be allowed as long as it is revelant to features specified in POSIX.1 {3}.

The test result code appropriate for each documentation assertion shall be determined. When the information specified by a documentation assertion does not appear in the designated section or sections, the result code for that assertion shall be **FAIL**.

Conformance documents, required by standard organizations, may be bound together as long as the POSIX.1 {3} part is clearly identifiable.

1.4.17.2 Recommendations

None.

1.4.18 PCD Announcement Mechanism

1.4.18.1 Requirements

When assertions are dependent on the PCD.1 for an announcement mechanism about the support or nonsupport of POSIX.1 {3} conditional features, the symbol to be used and its associated conditional feature are stated in Table 1.4.

Table 1.4—PCD Determinable Conditional Features

Symbol	Documentation
{PCD_CREAT_LINK_COUNT}	If supported, then creating a directory causes the link count of the directory in which it is created to be incremented.
{PCD_DIR_TYPE}	If supported, then type DIR is implemented as a file descriptor.
{PCD_LINK_TO_DIRECTORY}	If supported, then a calling process with the proper permission can create a link to a directory.
{PCD_LINK_FILE_SYSTEM}	If supported, then a link can be created between different file systems.
{PCD_NO_LOCK_FILE_TYPE}	If supported, then file types for which locking is unsupported do exist.
{PCD_READ_INTERRUPTED}	If supported, then *read*() when interupted by a signal after successfully reading some data returns the number of bytes read, otherwise returns an error value.
{PCD_WRITE_INTERRUPTED}	If supported, then *write*() when interupted by a. signal after successfully writing some data returns number of bytes written, otherwise returns an error value.
{PCD_WRITE_PERM_TO_RENAME}	If supported, then write permission is required for an existing directory to rename it.

1.4.18.2 Recommendations

None.

1.4.19 Test Methods for Common-Usage C Functions

1.4.19.1 Requirements

If an implementation claiming conformance to Common-Usage C does not support a C Standard {2} function or a PCTS.1 detects a deviation from the C Standard {2}, the corresponding assertions in 8.1.1 through 8.1.59 shall have

the result code **UNRESOLVED** or **FAIL**. If the audit of the PCD.1 shows that the PCD.1 declares the C function not implemented or the deviations are documented, the test result code is resolved to **PASS**; otherwise it remains **FAIL**. Additionally, if differences are not documented in the PCD.1, documentation assertion "DO1" in Section 8. shall have the result code **FAIL**.

This means that effectively each assertion in 8.1.1 through 8.1.59 (although not written that way) reads as follows:

XX(A) If the function is supported and there is no deviation from (assertion):
(assertion)

Otherwise:
Either the PCD.1 declares the C function not implemented, or the PCD.1 documents the deviation.

1.4.19.2 Recommendations

None.

2. Definitions and General Requirements

2.1 Conventions

This standard uses the following typographic conventions:

1) IEEE Std 1003.3-1991 {4} defines the nomenclature required for writing assertions. The description of the terms (A), (B), (C), (D), (Condition?A:B), R, GA, and the testing requirements are discussed in that standard.
2) When assertions are dependent on testing constraints, the nomenclature as described in 1.4.3.3 is used.
3) When assertions are dependent on PCD conditionals, the nomenclature as described in 1.4.3.4 is used.
4) Assertion lists generated for each POSIX.1 {3} element begin with a new clause or subclause. Each clause and subsequent subclauses contain a trailing POSIX.1 {3} subclause reference in parentheses.
5) The *italic* font is used for
   — The initial appearances of defined terms
   — Cross-references to defined terms within 1.4.2
   — Parameters (option arguments and operands) that are generally substituted with real values by the application
   — C-language data types and function names
   — Global external variable names
   — The trailing POSIX.1 {3} subclause reference of the element
6) The **bold** font is used for
   — Test result codes
7) The `constant-width` (Courier) font is used
   — To illustrate examples of system input or output where exact usage is depicted
   — For references to utility names, element groupings, and C-language headers
8) Symbolic *errno* names returned by functions are represented as [Symbolic_name].
9) Symbolic constant limits are represented as {Symbolic_constant_limit}.
10) Symbolic constant options are represented as {Option_name}.
11) Notes provided as parts of labeled tables and figures are integral parts of this standard (normative). Footnotes and notes within the body of the text are for information only (informative).
12) Defined names that are usually in lowercase, particularly function names, are never used at the beginning of a sentence or anywhere else that regular English usage would require them to be capitalized.
13) Mathematical symbols such as <, ≤, etc. are only used in formulas, assertion classification assignments, and conditional clauses preceding conditional assertions.

14) 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.
15) The double quote (“name”) is used when a name is specified.
16) The single quote (‘value’) is used when a specific value is specified.
17) The conventions listed above are for ease of reading only. Editorial inconsistencies in the use of typography are unintentional and have no normative meaning in this standard.

A summary of typographical conventions is shown in Table 2.1.

Table 2.1—Typographical Conventions

Reference	Example
C language header	`<sys/stat.h>`
Command name	`tar`
Data types	*long*
Defined terms	*file*
Environment variables	**PATH**
Error number	[EINTR]
File name	`filename`
Function argument declaration	extern unsigned long int
Function argument	*arg1*
Function declaration	long int
Function name	*funct*()
Global external	*errno*
IEEE Std 1003 .n-19xy reference	POSIX.n
Implementation-dependent limit	{MAX_INPUT}
Metacharacter	*(any character string)
Null test method	UNUSED
Parameters	<*path*>
Section x.y reference	See x.y or x.y
Special character	`<newline>`
Symbolic constant limit	{LINK_MAX}
Symbolic constant option	{_POSIX_JOB_CONTROL}
Symbolic constant value	{_POSIX_JOB_CONTROL}
Symbol defined in header	_POSIX_JOB_CONTROL
Table reference	Table 2.1
Variable	*st_atime*

2.2 Definitions

2.2.1 Terminology

There are no test methods provided for this subclause.

2.2.2 General Terms

NOTE — The subclause numbers that follow are not sequential because not all definitions have associated test assertions. Only those with specific test assertions are presented here.

2.2.2.4 Appropriate Privileges

D01(A) The means for associating the appropriate privileges with a process are described in 2.2.2.4 of the PCD.1 or in all the subclauses of the PCD.1 that correspond to subclauses in POSIX.1 {3} where privileged behavior is described.

2.2.2.9 Character Special File

D02(C) If character special files other than terminal device files are supported, and this is documented:
The details are contained in 2.2.2.9 of the PCD.1.

D03(C) If structures of character special files other than terminal device files are described:
The details are contained in 2.2.2.9 of the PCD.1.

2.2.2.27 File

D04(C) If file types other than regular file, character special file, block special file, FIFO special file, and directory are described:
The details are contained in 2.2.2.27 of the PCD.1.

2.2.2.30 File Group Class

D05(C) If the implementation uses additional criteria beyond those specified by POSIX.1 {3} to assign a process to the file group class of a file, and this is documented:
The details are contained in 2.2.2.30 of the PCD.1.

2.2.2.32 Filename

GA02 For *execl*(), *execle*(), *execv*(), *execve*(), *execlp*(), *execvp*(), *opendir*(), *chdir*(), *open*(), *creat*(), *link*() *existing*, *link*() *new*, *mkdir*(), *mkfifo*(), *unlink*(), *rmdir*(), *rename*() *old*, *rename*() *new*, *star*(), *access*(), *chmod*(), *chown*(), *utime*(), *pathconf*(), *fopen*(), *freopen*(), *remove*(), `tar` format creating utility, and `cpio` format creating utility:
An implementation supports *filenames* containing any of the following characters:
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ -

GA03 For *execl*(), *execle*(), *execv*(), *execve*(), *execlp*(), *execvp*(), *opendir*(), *chdir*(), *open*(), *creat*(), *link*() *existing*, *link*() *new*, *mkdir*(), *inkfifo*(), *unlink*(), *rmdir*(), *rename*() *old*, *rename*() *new*, *stat*(), *access*(), *chmod*(), *chown*(), *utirne*(), *pathconf*(), *fopen*(), *freopen*(), *remove*(), `tar` format creating utility, and `cpio` format creating utility:
A *filename* retains the unique identity of its upper- and lowercase letters.

01(B) No two directory entries in the same directory have the some name.

See Reason 3 in Section 5. of POSIX.3 {4}.

2.2.2.55 Parent Process ID

D06(A) The new parent process ID assigned to a process after the lifetime of its parent has ended is stated in 2.2.2.55 of the PCD.1.

2.2.2.57 Pathname

D07(A) Any special interpretation of a *pathname* that begins with exactly two slashes is stated in 2.2.2.57 of the PCD.1.

2.2.2.64 Process Group ID

02(B) A process group ID is not reused by the system until the process group lifetime ends.

See Reason 3 in Section 5. of POSIX. 3 {4}.

2.2.2.66 Process Group Lifetime

03(A) The lifetime of the process group ends when the last remaining process leaves the group.

Testing Requirements:

Test that the process group for the last remaining process in the process group that has been terminated but unwaited for still exists.

2.2.2.67 Process ID

04(B) A process that is not a system process does not have a process ID of 1.

See Reason 3 in Section 5. of POSlX.3 {4}.

05(B) A process ID is not reused by the system until the process lifetime ends.

See Reason 3 in Section 5. of POSIX.3 {4}.

06(B) A process ID is not reused while there exists a process group with a process group ID of the same number.

See Reason 3 in Section 5. of POSIX. 3 {4}.

2.2.2.68 Process Lifetime

D08(C) If documentation is provided that states the resources returned when a process terminates:

The details are contained in 2.2.2.68 of the PCD.1.

07(B) When a process completes a *wait*() or *waitpid*() for a process that is inactive due to termination, then all of the resources associated with the inactive process are returned to the system. The last of the resources returned is the process ID.

See Reason 3 in Section 5. of POSlX. 3 {4}.

2.2.2.69 Read-Only File System

D09(A) The specific restrictions on the semantics of system interfaces as they apply to objects on read-only file systems are contained in 2.2.2.69 of the PCD.1.

2.2.2.83 Supplementary Group ID

D10(C) If the effective group ID of a process is included in the list of supplementary group IDs returned by *getgroups*(), and this is documented:

The details are contained in 2.2.2.83 of the PCD.1.

2.3 General Concepts

2.3.1 Extended Security Controls

D01(C) If extended security controls are supported, and this is documented:
The extended security controls supported are stated in 2.3.1 of the PCD.1.

01(D) If the implementation supports extended security controls:
Extended security controls do not alter or override the defined semantics of any of the functions in POSIX.1 {3}.

See Reason 1 in Section 5. of POSIX.3 {4}.

2.3.2 File Access Permissions

D02(C) If whether additional or alternate file access control mechanisms are supported is documented:
The manner of support for additional or alternate file access control mechanisms or information that they are not supported is contained in 2.3.2 of the PCD.1.

GA04 For *funct*() of *execl*(), *execle*(), *execv*(), *execve*(), *execlp*(), *execvp*(), *open-dir*(), *chdir*(), *open*(), *creat*(), *link*() *existing*, *link*() *new*, *mkdir*(), *mkfifo*(), *unlink*(), *rmdir*(), *rename*() *old*, *rename*() *new*, *stat*(), *access*(), *chmod*(), *chown*(), *utime*(), and *pathconf*():
If the implementation provides the mechanism for creating processes with the appropriate privilege to override a file access control mechanism:
When the process has appropriate privileges for file access, and a call to function *funct*() requires read, write, or search access to the *pathname* argument, then the process is granted access to the file.

GA05 For *execl*(), *execle*(), *execv*(), *execve*(), *execlp*(), and *execvp*():
If the implementation provides the mechanism for creating processes with the appropriate privilege to override a file access control mechanism:
When the process has the appropriate privileges for file access, execute permission is granted to at least one user of the file, and a call to an `exec` type function requires execute access to the argument *path* or *file*, then the process is granted access.

GA06 For *funct*() of *execl*(), *execle*(), *execv*(), *execve*(), *execlp*(), *execvp*(), *open-dir*(), *chdir*(), *open*(), *creat*(), *link*() *existing*, *link*() *new*, *mkdir*(), *mkfifo*(), *unlink*(), *rmdir*(), *rename*() *old*, *rename*() *new*, *stat*(), *access*(), *chmod*(), *chown*(), *utime*(), and *pathconf*():
When the process does not have the required appropriate privileges to override the file access control mechanism, and the process requires read, write, execute, or search access to the pathname argument of function *funct*(), then a call to *funct*() is granted access in the case that the required access permission bit is set for the class (file owner class, file group class, or file other class) to which the process belongs.

GA07 If the implementation supports additional file access control mechanisms:
The mechanism only further restricts the access permissions defined by the file permission bits.

GA08 If the implementation supports alternate file access control mechanisms:
The mechanism specifies file permission bits for the file owner class, file group class, and file other class of the file corresponding to the access permissions to be returned by *stat*() and *fstat*().

GA09 If the implementation supports alternate file access control mechanisms:
The mechanism can be enabled only by explicit user action on a per-file basis by a process having the required appropriate privilege.

GA10 If the implementation supports alternate file access control mechanisms:
The mechanism can be enabled only by explicit user action on a per-file basis by the file owner.

GA11 If the implementation supports alternate file access control mechanisms:

The mechanism can be disabled for a file after the file permission bits are changed for that file with *chmod*().

D03(C) If both alternate and additional file access control mechanism(s) are provided, and whether disabling an alternate mechanism after *chmod*() disables additional mechanisms is documented:
The details are contained in 2.3.2 of the PCD.1.

2.3.3 File Hierarchy

There are no test methods provided for this subclause.

2.3.4 Filename Portability

There are no test methods provided for this subclause.

2.3.5 File Times Update

R01 For *close*():
When a file is closed by the last process that had it open, then all time-related fields marked for update are updated. [See Assertion 11 in 6.3.1.2.]

GA12 For *stat*() and *fstat*():
When *stat*() or *fstat*() is called for a file, then all time-related fields marked for update are updated, and time-related fields not marked for update are not updated.

D04(C) If an implementation of a function that is required by POSIX.1 {3} to update time-related fields for reasons other than pathname resolution updates fields other than those required:
The details are contained in 2.3.5 of the PCD.1 or in the subclause of the PCD.1 that corresponds to the *description* subclause in POSIX.1 {3} where the function is defined.

D05(C) If an implementation of a function that is not required by POSIX.1 {3} to update time-related fields happens to do so, and this is documented:
The details are contained in 2.3.5 of the PCD.1 or in the subclause of the PCD.1 that corresponds to the *description* subclause in POSIX.1 {3} where the function is defined.

D06(C) If whether the updating of the related fields is done immediately or periodically is documented:
The details are contained in 2.3.5 of the PCD.1 or in the subclause of the PCD.1 that corresponds to the *description* subclause in POSIX.1 {3} where the function is defined.

GA13 For *open*(), *creat*(), *link*() *existing*, *link*() *new*, *mkdir*(), *mkfifo*(), *unlink*(), *rmdir*(), *rename*() *old*, *rename*() *new*, *access*(), *chmod*(), *chown*(), and *utime*():
No time-related fields are updated on read-only file systems.

2.3.6 Pathname Resolution

NOTE — In each of the pathname resolution general assertions below, for the elements *rmdir*(), *rename*() new, and *unlink*(), the current working directory should be an empty directory in order to avoid the occurrence of avoidable error conditions. Some implementations will consider the attempt to remove the current working directory an error and will indicate this with the [EBUSY] error indication.

GA14 For *funct*() of *execl*(), *execle*(), *execv*(), *execve*(), *execlp*(), *execvp*(), *open-dir*(), *chdir*(), *open*(), *creat*(), *link*() *existing*, *link*() *new*, *mkdir*(), *mkfifo*(), *unlink*(), *rmdir*(), *rename*() *old*, *rename*() *new*, *stat*(), *access*(), *chmod*(), *chown*(), *utime*(), *pathconf*(), *fopen*(), *freopen*(), and *remove*():
When the first filename component of the pathname argument is " . ", and the pathname does not begin with a slash, then *funct*() resolves the pathname by locating the second filename component (when specified) in the current working directory.

GA15 For *funct*() of *execl*(), *execle*(), *execv*(), *execve*(), *execlp*(), *execvp*(), *open-dir*(), *chdir*(), *open*(), *creat*(), *link*() *existing*, *link*() *new*, *mkdir*(), *mkfifo*(), *unlink*(), *rmdir*(), *rename*() *old*, *rename*() *new*, *stat*(), *access*(), *chmod*(), *chown*(), *utime*(), *pathconf*(), *fopen*(), *freopen*(), and *remove*():

When the argument *path* or *file* points to the string "/", then *funct*() resolves the pathname to the root directory of the process.

GA16 For *funct*() of *execl*(), *execle*(), *execv*(), *execve*(), *execlp*(), *execvp*(), *open-dir*(), *chdir*(), *open*(), *creat*(), *link*() *existing*, *link*() *new*, *mkdir*(), *mkfifo*(), *unlink*(), *rmdir*(), *rename*() *old*, *rename*() *new*, *stat*(), *access*(), *chmod*(), *chown*(), *utime*(), *pathconf*(), *fopen*(), *freopen*(), and *remove*():

When the argument *path* or *file* points to the string "///", then *funct*() resolves the pathname to the root directory of the process.

GA17 For *funct*() of *execl*(), *execle*(), *execv*(), *execve*(), *execlp*(), *execvp*(), *open-dir*(), *chdir*(), *open*(), *creat*(), *link*() *existing*, *link*() *new*, *mkdir*(), *mkfifo*(), *unlink*(), *rmdir*(), *rename*() *old*, *rename*() *new*, *stat*(), *access*(), *chmod*(), *chown*(), *utime*(), *pathconf*(), *fopen*(), *freopen*(), and *remove*():

When the pathname argument points to a string beginning with a single slash or beginning with three or more slashes, then *funct*() resolves the pathname by locating the first filename component of the pathnome in the root directory of the process.

GA18 For *funct*() of *execl*(), *execle*(), *execv*(), *execve*(), *exedp*(), *execvp*(), *opendir*(), *chdir*(), *open*(), *creat*(), *link*() *existing*, *link*() *new*, *mkdir*(), *mkfifo*(), *unlink*(), *rmdir*(), *rename*() *old*, *rename*() *new*, *stat*(), *access*(), *chmod*(), *chown*(), *utime*(), *pathconf*(), *fopen*(), *freopen*(), and *remove*():

When the first filename component of the pathname argument is "..", the pathname does not begin with a slash, and the current working directory is not the root directory of the process, then *funct*() resolves the pathname by locating the second filename component (when specified) in the parent directory of the current working directory.

GA19 For *funct*() of *execl*(), *execle*(), *execv*(), *execve*(), *execlp*(), *execvp*(), *open-dir*(), *chdir*(), *open*(), *creat*(), *link*() *existing*, *link*() *new*, *mkdir*(), *mkfifo*(), *unlink*(), *rmdir*(), *rename*() *old*, *rename*() *new*, *stat*(), *access*(), *chmod*(), *chown*(), *utime*(), *pathconf*(), *fopen*(), *freopen*(), and *remove*():

When the argument *path* or *file* points to the string "`F1/`" and `F1` is a directory, then *funct*() resolves the pathname to `F1`, which is in the current working directory.

GA20 For *funct*() of *execl*(), *execle*(), *execv*(), *execve*(), *execlp*(), *execvp*(), *open-dir*(), *chdir*(), *open*(), *creat*(), *link*() *existing*, *link*() *new*, *mkdir*(), *mkfifo*(), *unlink*(), *rmdir*(), *rename*() *old*, *stat*(), *access*(), *chmod*(), *chown*(), *utime*(), *pathconf*(), *fopen*(), *freopen*(), and *remove*():

When the argument *path* or *file* points to the string "`F1//`" and `F1` is a directory, then *funct*() resolves the pathname to `F1`, which is in the current working directory.

For *rename*() new:

When the argument *new* points to the string "`F1/`/" and `F1` is an empty directory, then *rename*() resolves the pathname to `F1`, which is in the current working directory.

GA21 For *funct*() of *execl*(), *execle*(), *execv*(), *execve*(), *execlp*(), *execvp*(), *open-dir*(), *chdir*(), *open*(), *creat*(), *link*() *existing*, *link*() *new*, *mkdir*(), *mkfifo*(), *unlink*(), *rmdir*(), *rename*() *old*, *rename*() *new*, *star*(), *access*(), *chmod*(), *chown*(), *utime*(), *pathconf*(), *fopen*(), *freopen*(), and *remove*():

When the pathname argument points to the string "`F1/F2`", then *funct*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current, working directory.

GA22 For *funct*() of *execl*(), *execle*(), *execv*(), *execve*(), *execlp*(), *execvp*(), *open-dir*(), *chdir*(), *open*(), *creat*(), *link*() *existing*, *link*() *new*, *mkdir*(), *mkfifo*(), *unlink*(), *rmdir*(), *rename*() *old*, *rename*() *new*, *stat*(), *access*(), *chmod*(), *chown*(), *utime*(), *pathconf*(), *fopen*(), *freopen*(), and *remove*():

When the pathname argument points to the string "`F1/./F2`", then *funct*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory.

GA23 For *funct*() of *execl*(), *execle*(), *execv*(), *execve*(), *execlp*(), *execvp*(), *open-dir*(), *chdir*(), *open*(), *creat*(), *link*() *existing*, *link*() *new*, *mkdir*(), *mkfifo*(), *unlink*(), *rrndir*(), *rename*() *old*, *rename*() *new*, *stat*(), *access*(),*chmod*(), *chown*(), *utime*(), *pathconf*(), *fopen*(), *freopen*(), and *remove*():

When the pathname argument points to the string "F1/../F1/F2", then *funct*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory.

GA24 For *funct*() of *execl*(), *execle*(), *execv*(), *execve*(), *execlp*(), *execvp*(), *open-dir*(), *chdir*(), *open*(), *creat*(), *link*() *existing*, *link*() *new*, *mkdir*(), *mkfifo*(), *unlink*(), *rmdir*(), *rename*() *old*, *rename*() *new*, *stat*(), *access*(), *chmod*(), *chown*(), *utime*(), *pathconf*(), *fopen*(), *freopen*(), and *remove*():

When the pathname argument points to the string "`F1//F2`", then *funct*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory.

GA25 For *funct*() of *execl*(), *execle*(), *execv*(), *execve*(), *execlp*(), *execvp*(), *open-dir*(), *chdir*(), *open*(), *creat*(), *link*() *existing*, *link*() *new*, *mkdir*(), *mkfifo*(), *unlink*(), *rmdir*(), *rename*() *old*, *rename*() *new*, *stat*(), *access*(), *chmod*(), *chown*(), *utime*(), *pathconf*(), *fopen*(), *freopen*(), and *remove*():

If {_POSIX_NO_TRUNC} is not supported in the corresponding directory:

When the pathname component is a string of more than {NAME_MAX} bytes in a directory for which {_POSIX_NO_TRUNC}; is not supported, then *funct*() resolves the pathname component by truncating it to {NAME_MAX} bytes.

There are no assertions specific to this subclause.

2.4 Error Numbers

01(A) The element *errno* is defined and available to a process via

extern int errno;

02(A) When the header `<errno.h>` is included, then the error numbers [E2BIG], [EACCES], [EAGAIN], [EBADF], [EBUSY], [ECHILD], [EDEADLK], [EDOM], [EEXIST], [EFAULT], [EFBIG], [EINTR], [EINVAL], [EIO], [EISDIR], [EMFILE], [EMLINK], [ENAMETOOLONG], [ENFILE], [ENODEV], [ENOENT], [ENOEXEC], [ENOLCK], [ENOMEM], [ENOSPC], [ENOSYS], [ENOTDIR], [ENOTEMPTY], [ENOTTY], [ENXIO], [EPERM], [EPIPE], [ERANGE], [EROFS], [ESPIPE], [ESRCH], and [EXDEV] are defined, are nonzero, are distinct from each other, and can be represented in *errno.*

03(B) No function defined by POSIX.1 sets *errno* to zero to indicate an error.

See Reason 2 in Section 5. of POSIX.3 {4}.

D01(C) If the implementation supports additional errors not listed in 2.4, and this is documented:

The details are contained in 2.4 of the PCD.1 or in the subclause of the PCD.1 that corresponds to the errors subclause in POSIX.1 {3} for each interface that returns the error.

DGA02 For *fork*(), *execl*(), *execv*(), *execle*(), *execve*(), *execlp*(), *execvp*(), *sigaddset*(), *sigdelset*(), *sigismernber*(), *opendir*(), *readdir*(), *closedir*(), *getcwd*(), *access*(), *chown*(), *pathconf*(), *fpathconf*(), and *fcntl*():

If the implementation supports the detection of optional error conditions:

The details of the error conditions detected are contained in the subclause of the PCD.1 where the error values of the function are described.

GA26 For *fork*(), *execl*(), *execle*(), *execv*(), *execve*(), *execlp*(), *execvp*(), *sigaddset*(), *sigdelset*(), *sigismernber*(), *opendir*(), *readdir*(), *closedir*(), *access*(), *chown*(), and *fcntl*():

If the implementation does not support the detection of optional error conditions:

The associated action requested is successful (unless a different error condition is detected).

D02(A) The details concerning whether [EFAULT] is reliably detected are contained in 2.4 of the PCD.1 or in the subclause of the PCD.1 that corresponds to the *errors* subclause in POSIX.1 {3} for each interface that is documented optionally to return [EFAULT].

D03(A) The maximum file size allowed by the implementation is contained in 2.4 of the PCD.1.

2.5 Primitive System Data Types

01(A) When the header `<sys/types.h>` is included, then the data types *dev_t*, *gid_t*, *ino_t*, *mode_t*, *nlink_t*, *off_t*, *pid_t*, *size_t*, *ssize_t*, and *uid_t* are defined and are arithmetic data types.

02(A) The data type of each of *pid_t*, *off_t*, and *ssize_t* is signed arithmetic.

03(A) The data type of *size_t* is unsigned arithmetic.

04(A) The type *ssize_t* is capable of storing values in a range from "–1" to SSIZE_MAX.

05(B) When the header `<unistd.h>` is included, then the data types *size_t* and *ssize_t* are defined.

See Reason 1 in Section 5. of POSIX. 3 {4}.

D01(C) If additional type symbols ending in "_t" are defined in any header specified by POSIX.1 {3}, and this is documented:

The details are contained in 2.5 of the PCD.1.

2.6 Environment Description

01(C) If the environment variable **HOME** was defined by the implementation and currently has the value defined by the implementation:

The environment variable **HOME** corresponds to the *initial working directory* of the user from the user database.

02(D) If the environment variable **LOGNAME** was defined by the implementation and currently has the value defined by the implementation:

The environment variable **LOGNAME** corresponds to the *login name* associated with the current process.

NOTE — Testing that **LOGNAME** is composed of *characters from the portable filename character set* is not asserted since POSIX.1 {3} implies in a note that this condition should be tolerated.

See Reason 2 in Section 5. of POSIX.3 {4}.

GA27 For *execlp*() and *execvp*():

Only when the argument *file* does not contain a slash are the **PATH** prefixes used by *execlp*() and *execvp*().

GA28 For *execlp*() and *execvp*():

A colon ':' in **PATH** separates one path prefix from another. Each path prefix in **PATH** indicates a directory to be included in the search path.

GA29 For *execlp*() and *execvp*():

When a nonzero-length prefix is applied to a filename, then a '/' is inserted between the prefix and the filename.

GA30 For *execlp*() and *execvp*():

An occurrence of two adjacent colons '::' in **PATH** indicates that the current working directory is to be included in the search path.

GA31 For *execlp*() and *execvp*():

An initial ':' indicates the current working directory.

GA32 For *execlp*() and *execvp*():

A trailing ':' indicates the current working directory.

GA33 For *execlp*() and *execvp*():

The directories in the search path are searched in the order in which they occur in the **PATH** environment until an executable file of the specified name is found.

GA34 For *getenv*() and `exec` type functions:
Upper- and lowercase letters in the environment retain their unique identities.

GA35 For *getenv*() and `exec` type functions:
An implementation supports environment variable names consisting of characters in the portable filename character set.

D01(C) If the implementation supports the use of characters other than those in the portable character set in environment variable names, and this is documented:
The details are contained in 2.6 of the PCD.1.

2.7 C Language Definitions

2.7.1 Symbols From the C Standard

01(A) When the header `<time.h>` is included, then the types *clock_t* and *time_t* and the macro **NULL** are defined.

02(A) The type *clock_t* is capable of representing all integer values from zero to the number of clock ticks in 24 h.

03(A) When the header `<unistd.h>` is included, then the symbol **NULL** is defined and expands to either an integral constant expression with the value 0 or such an expression cast to type *(void *)*.

04(A) Each of the following headers `<dirent.h>`, `<fcntl.h>`, `<grp.h>`, `<pwd.h>`, `<setjmp.h>`, `<signal.h>`, `<stdio.h>`, `<sys/stat.h>`, `<sys/times.h>`, `<sys/wait.h>`, `<termios.h>`, `<time.h>`, `<utime.h>`, and `<unistd.h>` can be included more than once, in any combination of headers in any order, and a symbol can be defined with the same value in more than one of these headers.

NOTE — It is possible to include `<sys/types.h>` more than once, and the first instance of its inclusion precedes any other header that depends upon its prior inclusion.

2.7.2 POSIX.1 Symbols

NOTE — When {_POSIX_SOURCE} is defined before any header is included, then those symbols defined by POSIX.1 {3}; to be in the header are visible. (See 1.4.7.1.)

05(B) When {_POSIX_SOURCE} is defined and no other feature test macros are defined before any header is included, then no symbols other than those defined or reserved for that header by POSIX.1 {3} or the C Standard {2} are made visible. The constraints on the usage of reserved symbols are as specified in POSIX.1 {3}.

See Reason 5 in Section 5. of POSIX. 3 {4}.

D01(C) If the implementation supports feature test macros other than {_POSIX_SOURCE}, and this is documented:
The details are contained in 2.7.2 of the PCD.1.

2.7.2.1 C Standard Language-Dependent Support

There are no assertions specific to this subclause.

2.7.2.2 Common-Usage-Dependent Support

There are no assertions specific to this subclause.

2.7.3 Headers and Function Prototypes

GA36 For all elements with result type not *void* and not *int:*
If the implementation provides C Standard {2} support:

When the header <*.h> is included, then the function prototype *type1 funct(type2, type3)* is declared.[5]

Otherwise:

When the header <*.h> is included, then the function *funct*() is declared with the result type *type1*.

For all elements with result type *void* except *assert*():

If the implementation provides C Standard {2} support:

When the header <*.h> is included, then the function prototype *void funct(type2, type3*) is declared.

For all elements with result type *int except setjmp*() and *sigsetjmp*():

If the implementation provides C Standard {2} support:

When the header <*.h> is included, then the function prototype *int funct(type2, type3*) is declared.

Otherwise:

When the header <*.h> is included, then the function *funct*() is either declared with the result type *int* or is not declared in the header.

For *funct*() of *setjmp*() and *sigsetjmp*():

If *funct*() is not defined as a macro:

When the header <setjmp.h> is included, then *funct*() is declared as an identifier with external linkage and result type *int*.

GA37 If *funct*() is defined as a macro when the header <*.h> is included:

When the macro *funct*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), it then expands to an expression with the result type compatible with that specified for *funct*().

There are no assertions specific to this subclause.

2.8 Numerical Limits

2.8.1 C Language Limits

01(A) When the header <limits.h> is included, then the following symbols and corresponding values are defined and have values that meet the requirements of the C Standard {2}, as shown in Table 2.2.

[5]The function prototypes are aligned with ISO/IEC 9945-1 : 1990. When not specified in ISO/IEC 9945-1 : 1990, they are aligned with ISO/IEC 9899 :1990.

Table 2.2—C Language Limits

Symbol	Value	Description
CHAR_BIT	8	Minimum number of bits
CHAR_MAX		See note below
CHAR_MIN		See note below
INT_MAX	+32767	Minimum maximum
INT_MIN	-32767	Maximum minimum
LONG_MAX	+2147483647	Minimum maximum
LONG_MIN	-2147483647	Maximum maximum
MB_LEN_MAX	1	Minimum maximum
SCHAR_MAX	+127	Minimum maximum
SCHAR_MIN	-127	Maximum minimum
SHRT_MAX	+32767	Minimum maximum
SHRT_MIN	-32767	Maximum minimum
UCHAR_MAX	255	Minimum maximum
UINT_MAX	65535	Minimum maximum
ULONG_MAX	4294967295	Minimum maximum
USHRT_MAX	65535	Minimum maximum

NOTE — If *char* is signed, then {CHAR_MIN} is equal to {SCHAR_MIN} and {CHAR_MAX} is equal to {SCHAR_MAX}; otherwise, {CHARMIN} is zero and {CHAR_MAX} is equal to {UCHAR_MAX}.

2.8.2 Minimum Values

02(A) When the header `<limits.h>` is included, then the following symbols and corresponding values are defined as shown in Table 2.3.

Table 2.3—Minimum Values

Symbol	Value
{_POSIX_ARG_MAX}	4096
{_POSIX_CHILD_MAX}	6
{_POSIX_LINK_MAX}	8
{_POSIX_MAX_CANON}	255
{_POSIX_MAX_INPUT}	255
{_POSIX_NAME_MAX}	14
{_POSIX_NGROUPS_MAX}	0
{_POSIX_OPEN_MAX}	16
{_POSIX_PATH_MAX}	255
{_POSIX_PIPE_BUF}	512
{_POSIX_SSIZE_MAX}	32767
{_POSIX_STREAM_MAX}	8
{_POSIX_TZNAME_MAX}	3

R01 A conforming implementation provides values at least as large as those specified in the previous table for the related symbols of the conforming implementation. (See Assertions 4–15 in 2.8.4 and 2.8.5.)

2.8.3 Run-Time Increasable Values

D01(A) The magnitude limitation fixed for {NGROUPS_MAX} (defined in <limits.h>) is described in 2.8.3 of the PCD.1.

03(A) When the header <limits.h> is included, then the symbolic name {NGROUPS_MAX} is defined, and the value of the symbolic constant is not less than the value of {_POSIX_NGROUPS_MAX}.

2.8.4 Run-Time Invariant Values

04(C) If the symbolic constant {ARG_MAX} is defined:
When the header <limits.h> is included, then the value of {ARG_MAX} is not less than {_POSIX_ARG_MAX}.

05(C) If the symbolic constant {CHILD_MAX} is defined:
When the header <limits.h> is included, then the value of {CHILD_MAX} is not less than {_POSIX_CHILD_MAX}.

06(C) If the symbolic constant {OPEN_MAX} is defined:
When the header <limits.h> is included, then the value of {OPEN_MAX} is not less than {_POSIX_OPEN_MAX}.

07(C) If the symbolic constant {STREAM_MAX} is defined:
When the header <limits.h> is included, then the value of {STREAM_MAX} is not less than {_POSIX_STREAM_MAX}.

08(C) If the symbolic constant {STREAM_MAX} is defined:
When the header <limits.h> is included, then the value of {STREAM_MAX} is equal to the value of FOPEN_MAX in the header <stdio.h>.

09(C) If the symbolic constant {TZNAME_MAX} is defined:
When the header <limits.h> is included, then the value of {TZNAME_MAX} is not less than {_POSIX_TZNAME_MAX}.

D02(A) The run-time invariant values for {ARG_MAX}, {CHILD_MAX}, {OPEN_MAX}, {STREAM_MAX}, and {TZNAME_MAX} (defined in <limits.h>), are contained in 2.8.4 of the PCD.1.

2.8.5 Pathname Variable Values

10(C) If the symbolic constant {LINK_MAX} is defined:
When the header <limits.h> is included, then the value of {LINK_MAX} is not less than {_POSIX_LINK_MAX}.

11(C) If the symbolic constant {MAX_CANON} is defined:
When the header is included, then the value of {MAX_CANON} is not less than {_POSIX_MAX_CANON}.

12(C) If the symbolic constant {MAX_INPUT} is defined:
When the header <limits.h> is included, then the value of {MAX_INPUT} is not less than {_POSIX_MAX_INPUT}.

13(C) If the symbolic constant {NAME_MAX} is defined:
When the header <limits.h> is included, then the value of {NAME_MAX} is not less than {_POSIX NAME_MAX}.

14(C) If the symbolic constant {PATH_MAX} is defined:
When the header <limits.h> is included, then the value of {PATH_MAX} is not less than {_POSIX_PATH_MAX}.

15(C) If the symbolic constant {PIPE_BUF} is defined:
When the header <limits.h> is included, then the value of {PIPE_BUF} is not less than {_POSIX_PIPE_BUF}.

D03(A) The values for {LINK_MAX}, {MAX_CANON}, {MAX_INPUT}, {NAME_MAX}, {PATH_MAX}, and {PIPE_BUF} (defined in <limits.h>), conditions under which these values may change, and the limits of such variations, if any, are contained in 2.8.5 of the PCD.1.

2.8.6 Invariant Values

16(D) If the implementation provides a means of obtaining the value of {SSIZE_MAX} at runtime:
The runtime value of {SSIZE_MAX} is equal to the value of SSIZE_MAX defined in <limits.h>.

See Reason 3 in Section 5. of POSIX.3 {4}.

17(A) When the header <limits.h> is included, then the macro SSIZE_MAX is defined and evaluates to a value not less than {_PCSIX_SSIZE_MAX}.

2.9 Symbolic Constants

R01 The implementation has the header <unistd.h>. (See Assertions 1, 3, 5, and 6 in 2.9.1 through 2.9.4.)

D01(A) For constants {_POSIX_JOB_CONTROL} and {_POSIX_SAVED_IDS} (defined in <unistd.h>), the values, conditions under which these values may change, and the limits of such variations, if any, are contained in 2.9 of the PCD.1.

D02(C) If the constant {_POSIX_CHOWN_RESTRICTED} is defined in <unistd.h>:
The value of {_POSIX_CHOWN_RESTRICTED}, conditions under which this value may change, and the limits of such variations, if any, are contained in 2.9 of the PCD.1.

D03(C) If the constant {_POSIX_NO_TRUNC} is defined in <unistd.h>:
The value of {_POSIX_NO_TRUNC}, conditions under which this value may change, and the limits of such variations, if any, are contained in 2.9 of the PCD.1.

D04(C) If the constant {_POSIX_VDISABLE} is defined in <unistd.h>:
The value of {_POSIX_VDISABLE}, conditions under which this value may change, and the limits of such variations, if any, are contained in 2.9 of the PCD.1.

2.9.1 Symbolic Constants for the *access*() Function

01(A) When the header <unistd.h> is included, then the constants R_OK (test for read permission), W_OK (test for write permission), X_OK (test for execute or search permission), and F_OK (test for existence of file) are defined.

02(A) The constants F_OK, R_OK, W_OK, X_OK, and the expression R_OK|W_OK, R_OK|X_OK, W_OK|X_OK, and R_OK|W_OK|W_OK all have distinct integral values.

2.9.2 Symbolic Constants for the *lseek*() Function

03(A) When the header <unistd.h> is included, then the elements SEEK_SET, SEEK_CUR, and SEEK_END are defined.

04(A) The elements SEEK_SET, SEEK_CUR, and SEEK_END all have distinct integral values.

2.9.3 Compile-Time Symbolic Constants for Portability Specifications

R02 When the header `<unistd.h>` is included, then the value of the symbol {_POSIX_JOB_CONTROL} is not less restrictive than that provided by the respective POSIX option via *sysconf*(). (See Assertion 12 in 4.8.1.2.)

R03 When the header `<unistd.h>` is included, then the value of the symbol {_POSIX_SAVED_IDS} is not less restrictive than that provided by the respective POSIX option via *sysconf*(). (See Assertion 13 in 4.8.1.2.)

05(A) When the header `<unistd.h>` is included, then {_POSIX_VERSION} is defined and matches the integer value of POSIX.1 {3}, "199009L."

2.9.4 Execution-Time Symbolic Constants for Portability Specifications

D05(C) If {_POSIX_CHOWN_RESTRICTED} is not defined or defined with a value other than −1 in `<unistd.h>`:

The files and file types to which{_POSIX_CHOWN_RESTRICTED} applies are contained in 2.9.4 of the PCD.1.

R04 When the header `<unistd.h>` is included, and the value of the symbol {_POSIX_CHOWN_RESTRICTED} is defined and has a value other than −1, then the value of the symbol provides the same indication as that provided by the interrogation at run-time of the POSEC.1 {3} option via *pathconf*() and *fpathconf*(). (See Assertions 36 and 20 in 5.7.1.1.2 and 5.7.1.2.2.)

D06(C) If {_POSIX NO_TRUNC} is not defined or defined with a value other than −1 in `<unistd.h>`:

The files and file types to which {_POSIX_NO_TRUNC} applies are contained in 2.9.4 of the PCD.1.

R05 When the header `<unistd.h>` is included, and the value of the symbol {_POSIX_NO_TRUNC} is defined and has a value other than −1, then the value of the symbol provides the same indication as that provided by the interrogation at run-time of the POSIX.1 {3} option via *pathconf*() and *fpathconf*(). (See Assertions 38 and 22 in 5.7.1.1.2 and 5.7.1.2.2.)

D07(C) If {_POSIX_VDISABLE} is not defined or defined with value other than −1 in `<unistd.h>`:

The files and file types to which {_POSIX_VDISABLE} applies are contained in 2.9.4 of the PCD.1.

R06 When the header `<unistd.h>` is included, and the value of the symbol {_POSIX_VDISABLE} is defined and has a value other than −1, then the value of the symbol provides the same indication as that provided by the interrogation at run-time of the POSIX.1 {3} option via *pathconf*() and *fpathconf*(). (See Assertions 40 and 24 in 5.7.1.1.2 and 5.7.1.2.2.)

06(D) If {_POSIX_CHOWN_RESTRICTED} is defined with the value −1 when `<unistd.h>` is included:

When the header `<unistd.h>` is included and {_POSIX_CHOWN_RESTRICTED} is −1, then the implementation does not provide this option.

See Reason 3 in Section 5. of POSIX.3 {4}.

07(D) If {_POSIX_NO_TRUNC} is defined with the value −1 when `<unistd.h>` is included:

When the header `<unistd.h>` is included and {_POSIX_NO_TRUNC} is −1, then the implementation does not provide this option.

See Reason 3 in Section 5. of POSIX.3 {4}.

08(D) If {_POSIX_VDISABLE} is defined with the value −1 when `<unistd.h>` is included:

When the header `<unistd.h>` is included and {_POSIX_VDISABLE} is −1, then the implementation does not provide this option.

See Reason 3 in Section 5. of POSIX. 3 {4}.

3. Process Primitives

3.1 Process Creation and Execution

3.1.1 *fork*()

3.1.1.1 Synopsis

01(A) If the implementation provides C Standard {2}; support:

When the header `<unistd.h>` is included, then the function prototype `pid_t fork(void)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<unistd.h>` is included, then the function *fork*() is declared with the result type *pid_t*. (See GA36 in 2.7.3.)

02(C) If *fork*() is defined as a macro when the header `<unistd.h>` is included:

When the macro *fork*() is invoked, then it expands to an expression with the result type *pid_t*. (See GA37 in 2.7.3.)

03(C) If *fork*() is defined as a macro in the header `<unistd.h>`:

It protects its return value with extra parentheses when necessary. (See GA01 in 1.3.4.)

3.1.1.2 Description

04(A) When a call to *fork*() completes successfully, then the child process ID is different from the parent process ID.

05(B) When a call to *fork*() completes successfully, then the child process ID does not match any other active process ID.

See Reason 1 in Section 5. of POSIX.3 {4}.

06(B) When a call to *fork*() completes successfully, then the child process ID does not match any active process group ID.

See Reason 1 in Section 5. of POSIX.3 {4}.

07(A) When a call to *fork*() completes successfully, then the parent process ID of the child process is the process ID of the parent process.

08(A) When a call to *fork*() completes successfully, then each of the file descriptors for the child refers to the same open file description as the corresponding file descriptor of the parent.

09(A) When a call to *fork*() completes successfully, then the child process has its own copy of the open directory streams of the parent process.

10(A) When a call to *fork*() completes successfully, then the values of *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* for the child process are reset.

D01(C) If the implementation supports sharing open directory streams between parent and child processes, and this is documented:

The details are contained in 3.1.1.2 of the PCD.1.

11(A) When a call to *fork*() completes successfully, then file locks set by the parent are not inherited by the child.

12(A) When a call to *fork*() completes successfully, then pending alarms are cleared for the child process.

13(A) When a call to *fork*() completes successfully, then the set of signals pending for the child process is initialized to the empty set.

14(A) When a call to *fork*() completes successfully, then the current working directory and the root directory are the same as for the parent.

15(A) When a call to *fork*() completes successfully, then signal settings for the child process are the same as for the parent.

16(A) When a call to *fork*() completes successfully, then the real and effective user ID and group ID are the same as for the parent.

17(C) If the behavior associated with {_POSIX_SAVED_IDS} is supported:

When a call to *fork*() completes successfully, then the saved set-user ID and saved set-group ID are the same as for the parent.

18(A) When a call to *fork*() completes successfully, then the supplementary groups for the child are the same as for the parent.

19(A) When a call to *fork*() completes successfully, then the process group for the child is the same as for the parent.

20(PCTS_GTI_DEVICE?A:UNTESTED)

When a call to *fork*() completes successfully, then the controlling terminal for the child process is the same as for the parent.

21(A) When a call to *fork*() completes successfully, then the file mode creation mask is the same as for the parent.

22(A) When a call to *fork*() completes successfully, then the signal mask for the child process, is the same as for the parent.

23({_POSIX_JOB_CONTROL}?A:UNTESTED)

When a call to *fork*() completes successfully, then the child process is in the same session as its parent.

See Reason 1 in Section 5. of POSIX.3 {4}.

D02(C) If the implementation documents whether there is support for inheritance of process characteristics for *fork*() not defined in POSIX.1 {3}:

The details are contained in 3.1.1.2 of the PCD.1

24(A) When a call to *fork*() completes successfully, then both the parent and child process are capable of executing independently before either process terminates.

3.1.1.3 Returns

25(A) When a call to *fork*() completes successfully, then a value of (*pid_t*)0 is returned to the child process, and the child process IE, is returned to the parent process.

R01 When a call to *fork*() completes unsuccessfully, then a value of (*pid_t*)–1 is returned and sets *errno* to indicate the error. (See Assertions 26-28 in 3.1.1.4.)

3.1.1.4 Errors

26(B) When the system lacks the necessary resources to create another process, then a call to *fork*() returns a value of (*pid_t*)-1, sets *errno* to [EAGAIN], and no process is created.

See Reason 3 in Section 5. of POSIX.3 {4}.

27(PCTS_PROCESS_LIMIT?A:UNTESTED)

If {CHILD_MAX} ≤ {PCTS CHILD MAX}:

When the system-imposed limit on the total number of processes under execution by a single user {CHILD_MAX} is reached, then a call to *fork*() returns a value of (*pid_t*)-1, sets *errno* to [EAGAIN], and no process is created.

Otherwise:

{PCTS_CHILD_MAX} processes can be created.

D03(C) If the implementation supports the detection of [ENOMEM] for *fork*():

The details under which [ENOMEM] occurs for *fork*() are contained in 3.1.1.4 of the PCD.1. (See DGA02 in 2.4.)

28(B) If the implementation supports the detection of [ENOMEM] for *fork*():

When the process requires more space than the system is able to supply, then a call to *fork*() returns a value of (*pid_t*) –1, sets *errno* to [ENOMEM], and no process is created.

See Reason 1 in Section 5. of POSIX.3 {4}.

Otherwise:

When the process requires more space than the system is able to supply, then a call to *fork*() is successful (unless a different error condition is detected). (See GA26 in 2.4.)

See Reason 2 in Section 5. of POSIX.3 {4}.

3.1.2 *execl*(), *execle*(), *execv*(), *execve*(), *execlp*(), *execvp*()

NOTE — The term *exec* is a notation for each of the six functions; *execl*(), *execle*(), *execv*(), *execve*(), *execlp*(), and *execvp*().

3.1.2.1 Synopsis

01(A) For *execl*():

If the implementation provides C Standard {2} support:

When the header `<unistd.h>` is included, then the function prototype `int execl(const char *, const char *, …)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<unistd.h>` is included, then the function *execl*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

For *execv*():

If the implementation provides C Standard {2} support:

When the header `<unistd.h>` is included, then the function prototype `int execv(const char *, char * const [])` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<unistd.h>` is included, then the function *execv*() is either declared with result type int or not declared in the header. (See GA36 in 2.7.3.)

For *execle*():

If the implementation provides C Standard {2} support:

When the header `<unistd.h>` is included, then the function prototype `int execle(const char *, const char *,… )` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<unistd.h>` is included, then the function *execle*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

For *execve*():

If the implementation provides C Standard {2} support:

When the header `<unistd.h>` is included, then the function prototype `int execve(const char *, char * const [], char * const [ ])` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<unistd.h>` is included, then the function *execve*() is either declared with result type int or not declared in the header. (See GA36 in 2.7.3.)

For *execlp*():

If the implementation provides C Standard {2} support:

When the header <unistd.h> is included: then the function prototype `int execlp(const char *, const char *, ...)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header <unistd.h> is included, then the function *execlp*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

For *execvp*():

If the implementation provides C Standard {2} support:

When the header <unistd.h> is included, then the function prototype `int execvp(const char *, char * const [])` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header <unistd.h> is included, then the function *execvp*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) For *execl*():

If *execl*() is defined as a macro when the header <unistd.h> is included:

When the macro *execl*() is invoked with the correct argumenttypes (or compatible argument types in the case, that C Standard {2} support is provided), then it expands to an expression with the result type *int.* (See GA37 in 2.7.3.)

For *execv*():

If *execv*() is defined as a macro when the header <unistd.h> is included:

When the macro *execv*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int.* (See GA37 in 2.7.3.)

For *execle*():

If *execle*() is defined as a macro when the header <unistd.h> is included:

When the macro *execle*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type int. (See GA37 in 2.7.3.)

For *execve*():

If *execve*() is defined as a macro when the header <unistd.h> is included:

When the macro *execve*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int.* (See GA37 in 2.7.3.)

For *execlp*():

If *execlp*() is defined as a macro when the header <unistd.h> is included:

When the macro *execlp*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int.* (See GA37 in 2.7.3.)

For *execvp*():

If *execvp*() is defined as a macro when the header <unistd.h> is included:

When the macro *exevup*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int.* (See GA37 in 2.7.3.)

03(C) If an `exec` type function is defined as a macro in the header <unistd.h>:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 4.3.3.)

3.1.2.2 Description

04(A) There is no return from a successful call to an `exec` type function, and the calling process image is overlaid by the new process image.

05(A) The *argc* parameter to *main*() of the new process image is the count of the number of argument parameters passed (not including the terminating **NULL** pointer).

06(A) The *argv* parameter to *main*() of the new process image is a null-terminated array of character pointers to the arguments that were provided by the `exec` type function as the arguments for the new process image.

07(A) For *execle*() and *execve*():

The environment for the new process image is obtained from the argument *envp* (an array of character pointers to null-terminated strings that are terminated by a **NULL** pointer).

For *execl*(), *execv*(), *execlp*(), and *exevup*():

The environment for the new process image is obtained from the external variable *environ* in the calling process image.

08(A) The variable extern char ***environ* in the new process image is initialized as a pointer to an array of character pointers to the environment strings provided by the calling process. The array terminates with a **NULL** pointer.

09(A) For *execlp*() and *execvp*():

PATH prefixes are only used by *execlp*() and *exevup*() when the argument *file* does not contain a '/'. (See GA27 in 2.6.)

For *execl*(), *execv*(), *execle*(), and *execve*():

UNUSED.

10(A) For *execlp*() and *execvp*():

PATH prefixes are separated by a colon ':'. (See GA28 in 2.6.)

For *execl*(), *execv*(), *execle*(), and *execve*():

UNUSED.

1l(A) For *execlp*() and *execvp*():

When a nonzero-length prefix is applied to a filename, then a '/' is inserted between the prefix and the filename. (See GA29 in 2.6.)

For *execl*(), *execv*(), *execle*(), and *execve*():

UNUSED.

12(A) For *execlp*() and *execvp*():

When a **PATH** prefix of zero length is specified by '::', then this indicates the current working directory. (See GA30 in 2.6.)

For *execl*(), *execv*(), *execle*(), and *execve*():

UNUSED.

13(A) For *execlp*() and *execvp*():

When **PATH** begins with an initial ':', then this indicates the current working directory. (See GA31 in 2.6.)

For *execl*(), *execv*(), *execle*(), and *execve*():

UNUSED.

14(A) For *execlp*() and *execvp*():

When PATH has a trailing ':', then this indicates the current working directory. (See GA32 in 2.6.)

For *execl*(), *execv*(), *execle*(), and *execve*():

UNUSED.

15(A) For *execlp*() and *execvp*():

The **PATH** environment is searched from beginning to end, and each colon-separated element is used as a prefix to a filename until an executable program by the specified name is found. (See GA33 in 2.6.)

For *execl*(), *execv*(), *execle*(), and *execve*():

UNUSED.

16(A) Upper- and lowercase letters in environment variable names and corresponding assigned values retain their unique identities. (See GA34 in 2.6.)

17(A) Environment variable names support the portable filename character set. (See GA35 in 2.6.)

D01(A) The results of an *execlp*() or *execvp*() call in the search of directories in the absence of environment variable **PATH** is stated in 3.1.2.2 of the PCD.1.

18(A) If {ARG_MAX} ≤ {PCTS_ARG_MAX}:

The `exec` elements accept arguments including environment data of {ARG_MAX} bytes allowing for NULL terminators, pointers, or alignment bytes as defined in the implementation.

Otherwise:

The `exec` elements accept arguments including environment data of {PCTS_ARG_MAX} bytes allowing for NULL terminators, pointers, or alignment bytes as defined in the implementation.

19(A) If {ARG_MAX} ≤ {PCTS_ARG_MAX}:

When the number of bytes in the argument and environment lists are less than or equal to {ARG_MAX}, then the arguments are available in the corresponding *main*() arguments, and the environment is available from the external variable *environ.*

Otherwise:

When the number of bytes in the argument and environment lists are less than or equal to {PCTS_ARG_MAX), then the arguments are available in the corresponding *main*() arguments, and the environment is available from the external variable *environ.*

20(A) For *execl*(), *execle*(), and *execlp*():

The argument list referenced by *argv* in *main*() of the new process image is obtained from the arguments *arg0, arg1,…argn* in the calling process.

For *execv*(), *execve*(), and *execvp*():

The argument list referenced by *argv* in *main*() of the new process image is obtained from the argument *argv* in the calling process.

21(A) When the first filename component of the argument *path* or *file* is ".", and the pathname does not begin with a slash, then the `exec` type function resolves the pathname by locating the second filename component (when specified) in the current working directory. (See GA14 in 2.3.6.)

22(B) When the argument *path* or *file* points to the string "/", then the `exec` type function resolves the pathname to the root directory of the process. (See GA15 in 2.3.6.)

See Reason 2 in Section 5. of POSIX. 3 {4}.

23(B) When the argument *path* or *file* points to the string "///", then the `exec` type function resolves the pathname to the root directory of the process.(See GA16 in 2.3.6.)

See Reason 2 in Section 5. of POSIX.3 {4}.

24(A) When the *path* or *file* argument points to a string beginning with a single slash or beginning with three or more slashes, then the `exec` type function resolves the pathname by locating the first filename component of the pathname in the root directory of the process. (See GA17 in 2.3.6.)

25(A) When the first filename component of the *path* or *file* argument is "..", the pathname does not begin with a slash, and the current working directory is not the root directory of the process, then the `exec` type function

resolves the pathname by locating the second filename component (when specified) in the parent directory of the current working directory. (See GA18 in 2.3.6.)

26(B) When the argument *path* or *file* points to the string "F1/" and `F1` is a directory, then the `exec` type function resolves the pathname to F1, which is in the current working directory. (See GA19 in 2.3.6.)

See Reason 2 in Section 5. of POSIX.3 {4}.

27(B) When the argument *path* or *file* points to the string "F1//" and `F1` is a directory, then the `exec` type function resolves the pathname to `F1`, which is in the current working directory. (See GA20 in 2.3.6.)

See Reason 2 in Section 5. of POSIX. 3 {4}.

28(A) When the argument *path* or *file* points to the string "F1/F2", then the `exec` type function resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA21 in 2.3.6.)

29(A) When the argument *path* or *file* points to the string "F1/./F2", then the `exec` type function resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA22 in 2.3.6.)

30(A) When the argument *path* or *file* points to the string "F1/../F1/F2", then the `exec` type function resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA23 in 2.3.6.)

31(A) When the argument *path* or *file* points to the string "F1//F2", then the `exec` type function resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA24 in 2.3.6.)

32(C) If {_POSIX_NO_TRUNC} is not supported in the corresponding directory:

When the *path* or *file* argument pathname component is a string of more than {NAME_MAX} bytes in a directory for which {_POSIX_- NO_TRUNC} is not supported, then the `exec` type function resolves the pathname component by truncating it to {NAME_MAX} bytes. (See GA25 in 2.3.6.)

33(A) On a call to `exec` elements, the pathname supports *filenames* containing any of the following characters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ -

(See GA02 in 2.2.2.32.)

34(A) On a call to `exec` elements, the pathname retains the unique identity of its upper- and lowercase letters. (See GA03 in 2.2.2.32.)

35(A) For *execl*(), *execv*(), *execle*(), and *execve*():

When the function is granted execute access to *path* and is granted search access to the path prefix of *path*, then an [EACCES] error does not occur.

For *execlp*() and *execvp*():

When, after the appropriate element of **PATH** has been prefixed to *file*, the function is granted execute access to the resulting pathname, and when the function is granted search access to the path prefix of this pathname, then an [EACCES] error does not occur.

36(A) When file descriptors are opened in the calling process image, and they have the FD_CLOEXEC flag clear, then they remain open in the new process image. When they have the FD_CLOEXEC flag set, then they are closed.

37(A) When file descriptors remain open in the calling process image, then the file offset, file status, and file access mode of the open file description, as well as the file locks, remain unchanged by this call.

38(C) If {PCD_DIR_TYPE} is **TRUE** and the implementation supports the detection of [EBADF] for *closedir*() or *readdir*():

When a directory stream is open in the calling process image, then it is closed in the new process image.

39(A) When signals are set to signal-specific default)SIG_DFL) action in the calling process image, then they are set to SIG_DFL in the new process image.

40(A) When signals are set to ignore signal (SIG_IGN) in the calling process image, then they are set to SIG_IGN in the new process image.

41(A) When signals are set to be caught in the calling process image, then they are set to the default action in the new process image.

42(A) When the set-user ID mode of the file being executed is not set, then the effective group ID of the new process remains the same as in the calling process.

43(A) When the set-group ID mode of the file being executed is not set, then the effective group ID of the new process image remains the same as in the calling process.

44(PCTS_CHMOD_SET_IDS?A:UNTESTED)

When the set-user ID mode of the file being executed is set, then the effective user ID of the new process is set to the user ID of the file executed, and the real user ID remains unchanged.

45(PCTS_CHMOD_SET_IDS?A:UNTESTED)

When the set-group ID mode of the file being executed is set, then the effective group ID of the new process is set to the group ID of the file executed, and the real group ID and supplementary group IDs remain unchanged.

46(PCTS_CHMOD_SET_IDS?C:UNTESTED)

If the behavior associated with {_POSIX_SAVED_IDS} is supported:

The effective user ID and effective group ID of the new process image are saved for use by *setuid* ().

47(A) The new process image of the calling process inherits the following attributes from the calling process:

- Process ID
- Parent process ID
- Process group ID
- Session membership
- Real user ID
- Real group ID
- Supplementary groups ID
- Time left until alarm clock signal
- Current working directory
- Root directory
- File mode creation mask
- Process signal mask
- Pending signals
- *tms_utime*
- *tms_stime*
- *tms_cutime*
- *tms_cstime*

D02(C) If the implementation documents whether there is support for inheritance of process characteristics for `exec` type functions not defined in POSIX.1 {3}:

The details are contained in 3.1.2.2 of the PCD.1.

48(A) When a call to `exec` is successful, then the *st_atime* field of the new process image file is marked for update.

D03(C) If the implementation documents whether the *st_atime* field for a process image file is marked for update when the system is able to find the file but the `exec` type function fails:

The details are contained in 3.1.2.2 of the PCD.1.

3.1.2.3 Returns

R01 When a call to an `exec` element completes unsuccessfully, then a value of (*int*)−1 is returned and *errno* is set to indicate the error. Neither the *argv*[] and *envp*[] arrays of pointers nor the strings to which those array elements point are modified. File descriptors marked close-on-exec are not closed, signals set to be caught are not set to default; neither the effective user ID nor the effective group ID of the current process is changed; and the value of the global variable **environ**, the pointers it contains, or the strings to which they point are not modified. (See Assertions 49-60 in 3.1.2.4.)

3.1.2.4 Errors

49({ARG_MAX}≤{PCTS_ARG_MAX}?A:UNTESTED)

When the number of bytes used in the argument list and the environment, including NULL terminators, pointers, and alignment bytes, is greater than the system-imposed limit for `exec` ({ARG_MAX} bytes), then a call to these functions returns a value of *(int)*−1 and sets *errno* to [E2BIG]. Neither the *argv*[] and *envp*[] arrays of pointers nor the strings to which those array elements point are modified. File descriptors marked close-on-exec are not closed; signals set to be caught are not set to default; neither the effective user ID nor the effective group ID of the current process is changed; and the value of the global variable **environ**, the pointers it contains, or the strings to which they point are not modified.

50(A) When search permission is denied for a directory listed in the *path* prefix of the new process image file, then a call to `exec` returns a value of (*int*)−1 and sets *errno* to [EACCES]. Neither the *argv*[] and *envp*[] arrays of pointers nor the strings to which those array elements point are modified. File descriptors marked close-on-exec are not closed; signals set to be caught are not set to default; neither the effective user ID nor the effective group ID of the current process is changed; and the value of the global variable **environ**, the pointers it contains, or the strings to which they point are not modified.

51(A) When the new process image file denies execution permission, then a call to `exec` returns a value of (*int*)−1 and sets *errno* to [EACCES]. Neither the *argv*[] and *envp*[] arrays of pointers nor the strings to which those array elements point are modified. File descriptors marked close-on-exec are not closed; signals set to be caught are not set to default; neither the effective user ID nor the effective group ID of the current process is changed; and the value of the global variable **environ**, the pointers it contains, or the strings to which they point are not modified.

52(D) If there exists a type of file for which the implementation does not support execution of files of this type:

A call to `exec` with a file of that type returns a value of (*int*)-1 and sets *errno* to [EACCES]. Neither the *argv*[] and *envp*[] arrays of pointers nor the strings to which those array elements point are modified. File descriptors marked close-on-exec are not closed; signals set to be caught are not set to default; neither the effective user ID nor the effective group ID of the current process is changed; and the value of the global variable **environ**, the pointers it contains, or the strings to which they point are not modified.

See Reason 2 in Section 5. of POSIX.3 {4}.

D04(C) If the implementation allows the execution of irregular files, and this is documented:

The details are contained in 3.1.2.4 of the PCD.1.

53(A) For *execl*(), *execv*(), *execle*(), and *execve*():

If{PATH_MAX}≤{PCTS_PATH_MAX}:

When the length of the *path* argument exceeds the maximum number of characters in a pathname, then a call to `exec` returns a value of (*int*)−1 and sets *errno* to [ENAMETOOLONG]. Neither the *argv*[] and *envp*[] arrays of pointers nor the strings to which those array elements point are modified. File descriptors marked close-on-exec are not dosed; signals set to be caught are not set to default; neither the effective user ID nor the effective group ID of the current process is changed; and the value of the global variable **environ**, the pointers it contains, or the strings to which they point are not modified.

Otherwise:

When the length of the path argument is {PCTS_PATH_MAX}, then a call to `exec` is successful.

For *execlp*() and *execvp*():

If {PATH_MAX}≤{PCTS_PATH_MAX}:

When the length of each of the pathnames generated by prefixing an element of the environment variable **PATH** to the file argument exceeds the maximum number of characters in a pathname, then a call to `exec` returns a value of (*int*)-1 and sets *errno* to [ENAMETOOLONG]. Neither the *argv*[] and *envp*[] arrays of pointers nor the strings to which those array elements point are modified. File descriptors marked close-on-exec are not closed; signals set to be caught are not set to default; neither the effective user ID nor the effective group ID of the current process is changed; and the value of the global variable **environ**, the pointers it contains, or the strings to which they point are not modified.

Otherwise:

When the length of the file argument is {PCTS_PATH_MAX}, then a call to `exec` is successful.

54({NAME_MAX}≤{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported:

For *execl*(), *execv*(), *execle*(), and *execve*():

When the length of a pathname component is longer than the maximum number of bytes in a filename {NAME_MAX}, then a call to `exec` returns a value of (*int*)-1 and sets *errno* to [ENAMETOOLONG]. Neither the *argv*[] and *envp*[] arrays of pointers nor the strings to which those array elements point are modified. File descriptors marked close-on-exec are not closed; signals set to be caught are not set to default; neither the effective user ID nor the effective group ID of the current process is changed; and the value of the global variable **environ**, the pointers it contains, or the strings to which they point are not modified.

For *execlp*() and *execvp*():

When each of the pathnames generated by prefixing an element of the environment variable **PATH** to the file argument contains a pathname component that is longer than the maximum number of bytes in a filename {NAME_MAX}, then a call to `exec` returns a value of (*int*)-1 and sets *errno* to [ENAMETOOLONG]. Neither the *argv*[] and *envp*[] arrays of pointers nor the strings to which those array elements point are modified. File descriptors marked close-on-exec are not closed; signals set to be caught are not set to default; neither the effective user ID nor the effective group ID of the current process is changed; and the value of the global variable **environ**, the pointers it contains, or the strings to which they point are not modified.

55({NAME_MAX}>{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported:

For *execl*(), *execv*(), *execle*(), and *execve*():

When the length of a pathname component equals {PCTS_NAME_MAX}, then a call to an `exec` type function succeeds.

For *execlp*() and *execvp*():

When the length of all the pathnames generated by prefixing an element of the environment variable **PATH** to the file argument contain a pathname component of size equal to {PCTS_NAME_MAX}, then a call to an `exec` type function succeeds.

56(A) For *execl*(), *execv*(), *execle*(), and *execve*():

When a component of the *path* argument does not exist, then a call to `exec` returns a value of (*int*)-1 and sets *errno* to [ENOENT]. Neither the *argv*[] and *envp*[] arrays of pointers nor the strings to which those array elements point are modified. File descriptors marked close-on-exec are not closed; signals set to be caught are not set to default; neither the effective user ID nor the effective group ID of the current process is changed; and the value of the global variabl **environ**, the pointers it contains, or the strings to which they point are not modified.

For *execlp*() and *execvp*():

When none of the pathnames generated by prefixing an element of the environment variable **PATH** to the file argument exists, then a call to `exec` returns a value of (*int*)-1 and sets *errno* to [ENOENT]. Neither the *argv*[] and *envp*[] arrays of pointers nor the strings to which those array elements point are modified. File descriptors marked close-on-exec are not closed; signals set to be caught are not set to default; neither the effective user ID nor the effective group ID of the current process is changed; and the value of the global variable **environ**, the pointers it contains, or the strings to which they point are not modified.

57(A) When the *path* or *file* argument points to an empty string, then a call to `exec` returns a value of (*int*)-1 and sets *errno* to [ENOENT]. Neither the *argv*[] and *envp*[] arrays of pointers nor the strings to which those array elements point are modified. File descriptors marked close-on-exec are not closed; signals set to be caught are not set to default; neither the effective user ID nor the effective group ID of the current process is changed; and the value of the global variable **environ**, the pointers it contains, or the strings to which they point are not modified.

58(A) For *execl*(), *execv*(), *execle*(), and *execve*():

When a component of the *path* prefix of the new process image file is not a directory, then a call to `exec` returns a value of (*int*)`-1 and sets` *errno* to [ENOTDIR]. Neither the *argv*[] and *envp*[] arrays of pointers nor the strings to which those array elements point are modified. File descriptors marked close-on-exec are not closed; signals set to be caught are not set to default; neither the effective user ID nor the effective group ID of the current process is changed; and the value of the global variable **environ**, the pointers it contains, or the strings to which they point are not modified.

For *execlp*() and *execvp*():

When a component of the path prefix of each of the pathnames generated by prefixing an element of the environment variable **PATH** to the *file* argument is not a directory, then a call to `exec` returns a value of (*int*)-1 and sets *errno* to [ENOTDIR]. Neither the *argv*[] and *envp*[] arrays of pointers nor the strings to which those array elements point are modified. File descriptors marked close-on-exec are not closed; signals set to be caught are not set to default; neither the effective user ID nor the effective group ID of the current process is changed; and the value of the global variable **environ**, the pointers it contains, or the strings to which they point are not modified.

59(A) For *execl*(), *execv*(), *execle*(), and *execve*():

When the new process image file has the required permission but is not in the proper format, then a call to `exec` returns a value of (*int*)-1 and sets *errno* to [ENOEXEC]. Neither the *argv*[] and *envp*[] arrays of pointers nor the strings to which those array elements point are modified. File descriptors marked close-on-exec are not closed; signals set to be caught are not set to default; neither the effective user ID nor the effective group ID of the current process is changed; and the value of the global variable **environ,** the pointers it contains, or the strings to which they point are not modified.

For *execlp*() and *execvp*():

UNUSED.

D05(C) If the implementation supports the detection of [ENOMEM] for `exec`:

The conditions under which [ENOMEM] occurs for `exec` type functions is contained in 3.1.2.4 of the PCD.1. (See DGA02 in 2.4.)

60(B) If the implementation supports the detection of [ENOMEM] for `exec`:

When the new process image requires more memory than is allowed by the hardware or by system-imposed memory management constraints, then a call to `exec` returns a value of (*int*)-1 and sets *errno* to [ENOMEM]. Neither the *argv*[] and *envp*[] arrays of pointers nor the strings to which those array elements point are modified. File descriptors marked close-on-exec are not closed; signals set to be caught are not set to default; neither the effective user ID nor the effective group ID of the current process is changed; and the value of the global variable **environ**, the pointers it contains, or the strings to which they point are not modified.

See Reason 1 in Section 5. of POSIX.3 {4}.

Otherwise:

When the new process image requires more memory than is allowed by the hardware or by system-imposed memory management constraints, then a call to `exec` is successful (unless a different error condition is detected). (See GA26 in 2.4.)

See Reason 2 in Section 5. of POSIX.3 [4].

3.2 Process Termination

GA40 For *_exit*() and process termination by a signal:

The process ceases the execution of subsequent program statements.

For *exit*():

If the function *atexit*() is supported:

When the process returns from the functions previously registered by *atexit*(), then the process ceases the execution of subsequent program statements.

Otherwise:

The process ceases the execution of subsequent program statements.

For *abort*():

When the SIGABRT signal is not being caught or the signal handler associated with SIGABRT returns, then the process ceases the execution of subsequent program statements.

GA41 For *_exit*(), *exit*(), *abort*(), and process termination by a signal:

All open file descriptors are closed.

GA42 For *_exit*(), *exit*(), *abort*(), and process termination by a signal:

All open directory streams are closed.

GA43 For *_exit*() and process termination by a signal:

When the parent process of the calling process is waiting for this process, then it is notified of the calling process termination and the low-order 8 b of status are made available to it.

For *exit*():

If the function *atexit*() is supported:

When the process returns from the functions previously registered by *atexit*(), and the parent process of the calling process is waiting for this process, then it is notified of the calling process termination and the low-order 8 b of status are made available to it.

Otherwise:

When the process calls *exit*(), and the parent process of the calling process is waiting for this process, then it is notified of the calling process termination and the low-order 8 b of status are made available to it.

For *abort*():

When the SIGABRT signal is not being caught or the signal handler associated with SIGABRT returns, and when the parent process of the calling process is waiting for this process, then it is notified of the calling process termination and the low-order 8 b of status are made available to it.

GA44 For *_exit*() and process termination by a signal:

When the parent process of the calling process is not waiting for the process, then the exit status code is saved for return to the parent process whenever the parent process executes a subsequent *wait*() or *waitpid*() for this process.

For *exit*():

If the function *atexit*() is supported:

When the process returns from the functions previously registered by *atexit*(), and the parent process of the calling process is not waiting for the process, then the exit status code is saved for

return to the parent process whenever the parent process executes a subsequent *wait*() or *waitpid*() for this process.

Otherwise:

When the process calls *exit*(), and the parent process of the calling process is not waiting for the process, then the exit status code is saved for return to the parent process whenever the parent process executes a subsequent *wait*() or *waitpid*() for this process.

For *abort*():

When the SIGABRT signal is not being caught or the signal handler associated with SIGABRT returns, and when the parent process of the calling process is not waiting for the process, then the exit status code is saved for return to the parent process whenever the parent process executes a subsequent *wait*() or *waitpid*() for this process.

GA45 For *_exit*() and process termination by a signal:

When a parent process that is not a controlling process terminates without waiting for its child processes to terminate, then the remaining child processes are not terminated and are assigned a new parent process ID.

For *exit*():

If the function *atexit*() is supported:

When the process returns from the functions previously registered by *atexit*(), and a parent process that is not a controlling process terminates without waiting for its child processes to terminate, then the remaining child processes are not terminated and are assigned a new parent process ID.

Otherwise:

When the process calls *exit*(), and a parent process that is not a controlling process terminates without waiting for its child processes to terminate, then the remaining child processes are not terminated and are assigned a new parent process ID.

For *abort*():

When the SIGABRT signal is not being caught or the signal handler associated with SIGABRT returns, and when a parent process that is not a controlling process terminates without waiting for its child processes to terminate, then the remaining child processes are not terminated and are assigned a new parent process ID.

GA46 For *_exit*() and process termination by a signal:

If either the behavior associated with {_POSIX_JOB_CONTROL} is supported or SIGCHLD is supported:

When a process terminates, then a SIGCHLD signal is sent to its parent process.

For *exit*():

If the function *atexit*() is supported and either the behavior associated with {_POSIX_JOB_CONTROL} is supported or SIGCHLD is supported:

When the process returns from the functions previously registered by *atexit*(), then a SIGCHLD signal is sent to its parent process.

If the function *atexit*() is not supported and either the behavior associated with {_POSIX_JOB_CONTROL} is supported or SIGCHLD is supported:

When the process calls *exit*(), then a SIGCHLD signal is sent to its parent process.

For *abort*():

If either the behavior associated with {_POSIX_JOB_CONTROL} is supported or SIGCHLD is supported:

When the SIGABRT signal is not being caught or the signal handler associated with SIGABRT returns, then a SIGCHLD signal is sent to its parent process.

GA47 For *_exit*() and process termination by a signal:

When a controlling process terminates, then a SIGHUP signal is sent to each process in the foreground process group of the controlling terminal belonging to the controlling process. The

remaining child processes, which are not terminated upon delivery of the SIGHUP signal, are assigned a new parent process ID.

For *exit*():

If the function *atexit*() is supported:

When a controlling process returns from the functions previously registered by *atexit*(), then a SIGHUP signal is sent to each process in the foreground process group of the controlling terminal belonging to the controlling process. The remaining child processes, which are not terminated upon delivery of the SIGHUP signal, are assigned a new parent process ID.

Otherwise:

When a controlling process calls *exit*(), then a SIGHUP signal is sent to each process in the foreground process group of the controlling terminal belonging to the controlling process. The remaining child processes, which are not terminated upon delivery of the SIGHUP signal, are assigned a new parent process ID.

For *abort*():

When a controlling process terminates and the SIGABRT signal is not being caught or the signal handler associated with SIGABRT returns, then a SIGHUP signal is sent to each process in the foreground process group of the controlling terminal belonging to the controlling process. The remaining child processes, which are not terminated upon delivery of the SIGHUP signal, are assigned a new parent process ID.

GA48 For *_exit*() and process termination by a signal:

When the calling process, which is a controlling process, terminates, then the controlling terminal associated with the session is disassociated from the session.

For *exit*():

If the function *atexit*() is supported:

When the calling process, which is a controlling process, returns from the functions previously registered by *atexit*(), then the controlling terminal associated with the session is disassociated from the session.

Otherwise:

When the calling process calls *exit*() from a controlling process, then the controlling terminal associated with the session is disassociated from the session.

For *abort*():

When the SIGABRT signal is not being caught or the signal handler associated with SIGABRT returns and the calling process, which is a controlling process, terminates, then the controlling terminal associated with the session is disassociated from the session.

GA49 For *_exit*() and process termination by a signal:

If the behavior associated with {POSIX_JOB_CONTROL} is supported:

When termination of the process causes a process group to be orphaned, and any member of the newly orphaned process group is stopped, then a SIGHUP signal followed by a SIGCONT signal is sent to each process in the newly orphaned process group.

For *exit*():

If the function *atexit*() is supported and the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When the process returning from the functions previously registered by *atexit*() causes a process group to be orphaned, and any member of the newly orphaned process group is stopped, then a SIGHUP signal followed by a SIGCONT signal is sent to each process in the newly orphaned process group.

If the function *atexit*() is not supported and the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When the process calling *exit*() causes a process group to be orphaned, and any member of the newly orphaned process group is stopped, then a SIGHUP signal followed by a SIGCONT signal is sent to each process in the newly orphaned process group.

For *abort*():

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When the SIGABRT signal is not being caught or the signal handler associated with SIGABRT returns, and when termination of the process causes a process group to be orphaned, and when any member of the newly orphaned process group is stopped, then a SIGHUP signal followed by a SIGCONT signal is sent to each process in the newly orphaned process group.

GA50 For *abort*():

When the SIGABRT signal is not being caught or the signal handler associated with SIGABRT returns, then on a call to *abort*() the status made available to *wait*() or *waitpid*() is that of a process terminated by the SIGABRT signal.

There are no assertions specific to this subclause.

3.2.1 Wait for Process Termination

3.2.1.1 *wait*() (3.2.1)

3.2.1.1.1 Synopsis (3.2.1.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<sys/wait.h>` is included, then the function prototype `pid_t wait(int *)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<sys/wait.h>` is included, then the function *wait*() is declared with the result type *pid_t*. (See GA36 in 2.7.3.)

02(C) If *wait*() is defined as a macro when the header `<sys/wait.h>` is included:

When the macro *wait*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *pid_t*. (See GA37 in 2.7.3.)

03(C) If *wait*() is defined as a macro in the header `<sys/wait.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

3.2.1.1.2 Description (3.2.1.2)

D01(C) If the order in which status is reported when status information is available for two or more child processes is documented:

The details are contained in 3.2.1.2 of the PCD.1.

04(A) A call to *wait*() suspends execution of the calling process until status information for one of its terminated child processes is available or until delivery of a signal whose action is either to execute a signal-catching function or to terminate the calling process.

05(A) When status information is available prior to a call to *wait*(), then return is immediate.

06(A) When the status of a child is or becomes available, then the value returned by *wait*() is the value of the process ID of the child process.

Testing Requirements:

Test with *stat_loc* set to NULL and *stat_loc* set to a valid address.

R01 When the argument *stat_loc* is not NULL then on a successful return from *wait*() the status information is stored in the location pointed to by *stat_loc*. (See Assertions 9, 11, and 12 in 3.2.1.1.2.)

07(A) When the status returned is from a terminated child process that returned a value of zero from *main*(), or when a value of zero is passed as the status argument to *_exit*() or *exit*(), then the value stored at the location pointed to by *stat_loc* is zero.

08(A) When the header `<sys/wait.h>` is included, the macros WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG, WIFSTOPPED, and WSTOPSIG are defined.

09(A) When the header `<sys/wait.h>` is included and the stratus is returned for a child process that either returned from *main*(), called *_exit*(), or called *exit*(), then the macro WIFEXITED(**stat_loc*) evaluates to a nonzero value, and the macros WIFSIGNALED(**stat_loc*) and WIFSTOPPED(**stat_loc*) evaluate to zero.

10(A) When the header `<sys/wait.h>` is included and the value of WIFEXITED(**stat_loc*) is nonzero—where *stat_loc* is the status argument populated by an invocation of *wait*()—then the macro WEXITSTATUS(**stat_loc*) evaluates to the low-order 8 b of the status argument that the child process passed to *_exit*() or *exit*(), or to the low-order 8 b of the value the child process returned from *main*().

11(A) When the header `<sys/wait.h>` is included and the status was returned for a child process that terminated due to the receipt of a signal that was not caught, then the macro WIFSIGNALED(**stat_loc*) evaluates to a nonzero value, and the macros WIFEXITED(**stat_loc*) and WIFSTOPPED(**stat_loc*) evaluate to zero.

12(A) When the header `<sys/wait.h>` is included and the value of WIFSIGNALED(**stat_loc*) is nonzero—where *stat_loc* is the status argument populated by an invocation of *wait*()—then the macro WTERMSIG(**stat_loc*) evaluates to the number of the signal that caused the termination of the child process.

13(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

A call to *wait*() does not return when one of the child processes of the calling process is stopped.

D02(C) If an implementation supports additional circumstances under which *wait*() or *waitpid*() report status, and this is documented:

The details on the circumstances and the meaning of the reported status bytes are contained in 3.2.1.2 of the PCD.1.

3.2.1.1.3 Returns (3.2.1.3)

R02 When a call to *wait*() completes successfully, then the value returned is equal to the process ID of the child process whose status is reported. (See Assertion 6 in 3.2.1.1.2.)

R03 When the delivery of a signal causes *wait*() to complete unsuccessfully, *wait*() returns a value of (*pid_t*)-1 and sets *errno* to [EINTR]. (See Assertion 15 in 3.2.1.1.4.)

R04 When a call to *wait*() completes unsuccessfully, then a value of (*pid_t*)–1 is returned and sets *errno* to indicate the error. (See Assertions 14 and 15 in 3.2.1.1.4.)

3.2.1.1.4 Errors (3.2.1.4)

14(A) When the calling process has no existing unwaited-for child processes, then a call to *wait*() returns a value of (*pid_t*)-1 and sets *errno* to [ECHILD].

15(A) When a call to *wait*() is interrupted by a signal, then *wait*() returns a value of (*pid_t*)-1 and sets *errno* to [EINTR]. The wait statuses of all child processes of the calling process remain available for subsequent calls to *wait*() or *waitpid*().

D03(C) If the value of the location pointed to by *stat_loc* when a call to *wait*() or *waitpid*() returns an *errno* of [EINTR] is documented:

The details are contained in 3.2.1.4 of the PCD.1.

3.2.1.2 *waitpid*() (3.2.1)

3.2.1.2.1 Synopsis (3.2.1.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<sys/wait.h>` is included, then the function prototype `pid_t waitpid(pid_t, int *, int)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<sys/wait.h>` is included, then the function *waitpid*() is declared with the result type *pid_t*. (See GA36 in 2.7.3.)

02(C) If *waitpid*() is defined as a macro when the header `<sys/wait .h>` is included:

When the macro *waitpid*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *pid_t*. (See GA37 in 2.7.3.)

03(C) If *waitpid*() is defined as a macro in the header `<sys/wait.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

3.2.1.2.2 Description (3.2.1.2)

04(A) A call to *waitpid*(*-1*, *stat_loc*, *0*) suspends execution of the calling process until status information for any one of its terminated child processes is available or until delivery of a signal whose action is either to execute a signal-catching function or to terminate the calling process.

05(A) When status information for any child of the calling process is available prior to a call to *waitpid*(*-1*, *stat_loc*, *options*), then return is immediate.

06(A) When the status of any child of the calling process is or becomes available, then the value returned by *waitpid*(*-1*, *stat_loc*, *options*) is the value of the process ID of the child process.

Testing Requirements:

Test with *stat_loc* set to NULL and *stat_loc* set to a valid address.

07(A) When the argument *pid* is greater than zero and is equal to the process ID of a child of the calling process, then a call to *waitpid*(*pid*, *stat_loc*, *options*) returns when the process specified by *pid* has terminated.

08(A) When the argument *pid* is equal to zero, then a call to *waitpid*(*pid*, *stat_loc*, *options*) returns when any child process that is a member of the same process group as the calling process has terminated.

09(A) When the argument *pid* is less than –1, then a call to *waitpid*(*pid*, *stat_loc*, *options*) returns when any child process with a process group ID equal to the absolute value of the argument *pid* has terminated.

10(A) When the header `<sys/wait.h>` is included, then the flags WNOHANG and WUNTRACED are defined as bitwise exclusive integer values.

11(A) When status is not immediately available for one of the child processes specified by *pid*, and the options argument specifies the WNOHANG flag, then a call to *waitpid*(*pid*, *stat_loc*, *options*) does not suspend execution of the calling process, and a value of zero is returned.

Testing Requirements:

If {_POSIX_JOB_CONTROL} is supported, test with options set to WNOHANG | WUNTRACED in addition to testing with options set to WNOHANG.

12(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

A call to *waitpid*(*pid*, *stat_loc*, *WUNTRACED*) or to *waitpid*(*pid*, *stat_loc*, *WNOHANG* | *WUNTRACED*) reports to the calling process the status of any child process specified by *pid* that is stopped and has not been reported since it stopped, or the status of any child process that has exited.

R01 When the argument *stat_loc* is not NULL, then on a successful return from a call to *waitpid*(*pid, stat_loc, options*), the status information is stored in the location pointed to by *stat_loc*. (See Assertions 14, 16, and 17 in 3.2.1.2.2.)

13(A) When the status returned to *waitpid*(*pid, stat_loc, options*) is from a terminated child process that returned a value of zero from *main*() or passed a value of zero as the status argument to *_exit*() or *exit*(), then the value stored at the location pointed to by *stat_loc* is zero.

14(A) When the header `<sys/wait.h>` is included and the status is returned for a child process by returning from *main*() or by a call to *_exit*() or *exit*(), then the macro WIFEXITED(**stat_loc*) evaluates to a nonzero value, and the macros WIFSIGNALED(**stat_loc*) and WIFSTOPPED(**stat_loc*) evaluate to zero.

15(A) When the header `<sys/wait.h>` is included and the value of WIFEXITED(**stat_loc*) is nonzero, then the macro WEXITSTATUS(**stat_loc*) evaluates to the low-order 8 b of the status argument that the child process passed to *exit*() or *exit*(), or to the low-order 8 b of the value the child process returned from *main*().

16(A) When the header `<sys/wait.h>` is included and the status was returned for a child process that terminated due to the receipt of a signal that was not caught, then the macro WIFSIGNALED(**stat_loc*) evaluates to a nonzero value, and the macros WIFEXITED(**stat_loc*) and WIFSTOPPED(**stat_loc*) evaluate to zero.

17(A) When the header `<sys/wait.h>` is included and the value of WIFSIGNALED(**stat_loc*) is nonzero, then the macro WTERMSIG(**stat_loc*) evaluates to the number of the signal that caused the termination of the child process.

18(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:
When the header `<sys/wait.h>` is included and the status was returned for a child process that is currently stopped, then the macro WIFSTOPPED(**stat_loc*) evaluates to a nonzero value, and the macros WIFEXITED(**stat_loc*) and WIFSIGNALED(**stat_loc*) evaluate to zero.

19(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:
When the WUNTRACED flag is clear, then a call to *waitpid*() does not return the status of its stopped child processes.

20(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:
When the header `<sys/wait.h>` is included and the value of WIFSTOPPED(**stat_loc*) is nonzero, then the macro WSTOPSIG(**stat_loc*) evaluates to the number of the signal that caused the process to stop.

3.2.1.2.3 Returns (3.2.1.3)

R02 When a call to *waitpid*() completes successfully, then the value returned is equal to the process ID of the child process whose status is reported. (See Assertion 6 in 3.2.1.2.2.)

R03 When the delivery of a signal causes *waitpid*() to complete unsuccessfully, *waitpid*() returns a value of (*pid_t*)–1 and sets *errno* to [EINTR]. (See Assertion 26 in 3.2.1.2.4.)

R04 When status is not immediately available for one of the child processes specified by pid and the options argument specifies the WNOHANG flag, then a call to *waitpid*(*pid, stat_loc, options*) does not suspend execution of the calling process and a value of zero is returned. (See Assertion 11 in 3.2.1.2.4.)

R05 When a call to *waitpid*() completes unsuccessfully, then a value of (*pid_t*)–1 is returned and sets *errno* to indicate the error. (See Assertions 21-27 in 3.2.1.2.4.)

3.2.1.2.4 Errors (3.2.1.4)

21(A) When the process or process group specified by *pid* does not exist, then a call to *waitpid*(*pid, stat_loc, options*) returns a value of (*pid_t*)–1 and sets *errno* to [ECHILD]. The wait statuses of all child processes of the calling process remain available for subsequent calls to *wait*() or *waitpid*().

22(A) When *pid* is equal to −1 and the calling process does not have any child processes, then a call to *waitpid*(*pid, stat_loc, options*) returns a value of (*pid_t*)−1 and sets *errno* to [ECHILD].

23(A) When pid is equal to 0 and the calling process does not hav any child processes in its process group, then a call to *waitpid*(*pid, stat_loc, options*) returns a value of (*pid_t*)−1 and sets *errno* to [ECHILD]. The wait statuses of all child processes of the calling process remain available for subsequent calls to *wait*() or *waitpid*().

24(A) When *pid* is less than −1 and the calling process does not have any child processes in the process group specified by pid, then a call to *waitpid*(*pid, stat_loc, options*) returns a value of (*pid_t*)−1 and sets *errno* to [ECHILD].

25(A) When *pid* is greater than zero and the process *pid* is not a child of the calling process, then a call to *waitpid*(*pid, stat_loc, options*) returns a value of (*pid_t*)−1 and sets *errno* to [ECHILD]. The wait status of the process *pid* remains available for subsequent calls to *wait*() or *waitpid*() by its parent process.

26(A) When a call to *waitpid*(*pid, stat_loc, options*) is interrupted by a signal, then *waitpid*() returns a value of (*pid_t*)−1 and sets *errno* to [EINTR]. The wait statuses of all child processes of the calling process remain available for subsequent calls to *wait*() or *waitpid*().

27(A) When the value of the options argument is not valid then a call to *waitpid*(*pid, stat_loc, options*) returns a value of (*pid_t*)−1 and sets *errno* to [EINVAL]. The wait statuses of all child processes of the calling process remain available for subsequent calls to *wait*() or *waitpid*().

3.2.2 *_exit*()

3.2.2.1 Synopsis

01(C) If the implementation provides C Standard {2} support:

When the header `<unistd.h>` is included, then the function prototype `void _exit(int)` is declared. (See GA36 in 2.7.3.)

D01(C) If the implementation provides Common-Usage C support:

The result type for function *_exit*() is contained in 3.2.2.1 of the PCD.1.

02(D) If the implementation provides C Standard {2} support and *_exit*() is defined as a macro when the header `<unistd.h>` is included:

When the macro *_exit*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type void. (See GA37 in 2.7.3.)

See Reason 2 in Section 5. of POSIX.3 {4}.

03(C) If *_exit*() is defined as a macro in the header `<unistd.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary. (See GA01 in 1.3.3.)

3.2.2.2 Description

04(A) On a call to *_exit*(), the process ceases the execution of subsequent program statements. (See GA40 in 3.2.)

05(A) On a call to *_exit*(), all open file descriptors are closed. (See GA41 in 3.2.)

06(B) On a call to *_exit*(), all open directory streams are closed. (See GA42 in 3.2.)

See Reason 2 in Section 5. of POSIX.3 {4}.

07(A) When the parent process of the calling process is executing a *wait*() or *waitpid*(), then on a call to *_exit*() by the child, the parent is notified of the calling process termination, and the low-order 8 b of status are made available to the parent. (See GA43 in 3.2.)

08(A) When the parent process of the calling process is not waiting for the process, then on a call to *_exit*() the exit status code is saved for return to the parent process whenever the parent process executes a subsequent *wait*() or *waitpid*() for this process. (See GA44 in 3.2.)

09(A) When a parent process that is not a controlling process terminates by calling *_exit*() without waiting for its child processes to terminate, then the remaining child processes are not terminated and are assigned a new parent process ID. (See GA45 in 3.2.)

D02(A) The new parent process ID assigned to children of a terminated process is stated in 3.2.2 of the PCD.1.

10(C) If either the behavior associated with {_POSIX_JOB_CONTROL} is supported or SIGCHLD is supported:

When a process terminates by calling *_exit*() a SIGCHLD signal is sent to its parent process. (See GA46 in 3.2.)

11(PCTS_GTI_DEVICE?A:UNTESTED)

When a process that is a controlling process terminates by calling *_exit*(), then a SIGHUP signal is sent to each process in the foreground process group of the controlling terminal belonging to the controlling process. The remaining child processes, which are not terminated upon delivery of the SIGHUP signal, are assigned a new parent process ID. (See GA47 in 3.2.)

12(PCTS_GTI__DEVICE?A:UNTESTED)

When the calling process is a controlling process, process termination causes the controlling terminal associated with the session to be disassociated from the session. (See GA48 in 3.2.)

13(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When termination of the process causes a process group to be orphaned, and any member of the newly orphaned process group is stopped, then a SIGHUP signal followed by a SIGCONT signal is sent to each process in the newly orphaned process group. (See GA49 in 3.2.)

3.2.2.3 Returns

R01 A call to *_exit*() does not return. (See Assertions 4–13 in 3.2.2.2.)

3.3 Signals

3.3.1 Signal Concepts

3.3.1.1 Signal Names

01(A) When the header `<signal.h>` is included, then the *sigset_t* type and the structure *sigaction* are declared, and the symbolic constants SIG_DFL and SIG_IGN are defined.

02(A) The constants SIG_DFL and SIG_IGN are of a type compatible with *void(*)()*.

03(B) The values of SIG_DFL and SIG_IGN match no declarable function.

See Reason 3 in Section 5. of POSIX.3 {4}.

04(A) When the header `<signal.h>` is included, then the symbolic signal constants SIGABRT, SIGALRM, SIGFPE, SIGHUP, SIGILL, SIGINT, SIGKILL, SIGPIPE, SIGQUIT, SIGSEGV, SIGTERM, SIGUSR1, SIGUSR2, SIGCHLD, SIGCONT, SIGSTOP, SIGTSTP, SIGTTIN, and SIGTTOU are defined.

05(A) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

The signals SIGABRT, SIGALRM, SIGFPE, SIGHUP, SIGILL, SIGINT, SIGKILL, SIGPIPE, SIGQUIT, SIGSEGV, SIGTERM, SIGUSR1, SIGUSR2, SIGCHLD, SIGCONT, SIGSTOP, SIGTSTP, SIGTTIN, and SIGTTOU have distinct positive integral values.

Otherwise:

The signals SIGABRT, SIGALRM, SIGFPE, SIGHUP, SIGILL, SIGINT, SIGKILL, SIGPIPE, SIGQUIT, SIGSEGV, SIGTERM SIGUSR1, and SIGUSR2 have distinct positive integral values.

06(A) The default action for the signals SIGABRT, SIGALRM, SIGFPE, SIGHUP, SIGILL, SIGINT, SIGKILL, SIGPIPE, SIGQUIT, SIGSEGV, SIGTERM, SIGUSR1, and SIGUSR2 is abnormal termination of the process.

07(A) When a signal is received whose action is to terminate the process, then the process is terminated. (See GA40 in 3.2.)

08(A) When a signal is received whose action is to terminate the process, then all open file descriptors in the process are closed. (See GA41 in 3.2.)

09(B) When a signal is received whose action is to terminate the process, then all open directory streams in the process are closed. (See GA42 in 3.2.)

See Reason 2 in Section 5. of POSIX.3 {4}.

10(A) When the parent process of the process terminated by a signal is waiting for the process, then it is notified of the calling process termination, and the signal number that caused the termination is made available to it. (See GA43 in 3.2.)

11(A) When the parent process of the process terminated by a signal is not waiting for the process, then the signal number that caused the termination is saved for return to the parent process whenever the parent process executes a subsequent *wait*() or *waitpid*() for this process. (See GA44 in 3.2.)

12(A) When a parent process that is not a controlling process terminates without waiting for its child processes to terminate, then the remaining child processes are not terminated and are assigned a new parent process ID. (See GA45 in 3.2.)

13(C) If either the behavior associated with {_POSIX_JOB_CONTROL} is supported or SIGCHLD is supported:
When a process terminates, a SIGCHLD signal is sent to its parent process. (See GA46 in 3.2.)

14(PCTS_GTI_DEVICE?A:UNTESTED)

When a process that is a controlling process terminates, then a SIGHUP signal is sent to each process in the foreground process group of the controlling terminal belonging to the controlling process. The remaining child processes, which are not terminated upon delivery of the SIGHUP signal, are assigned a new parent process ID. (See GA47 in 3.2.)

15(PCTS_GTI_DEVICE?A:UNTESTED)

When the calling process is a controlling process, process termination causes the controlling terminal associated with the session to be disassociated from the session. (See GA48 in 3.2.)

16(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:
When termination of the process causes a process group to be orphaned, and any member of the newly orphaned process group is stopped, then a SIGHUP signal followed by a SIGCONT signal is sent to each process in the newly orphaned process group. (See GA49 in 3.2.)

17(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:
The default action for SIGCHLD is to ignore the signal.

18(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:
The default action for SIGCONT is to continue the process if it is stopped; otherwise, the default action is to ignore the signal.

19({POSIX_JOB_CONTROL}?C:UNTESTED)

If SIGSTOP is supported:
The default action for SIGSTOP is to stop the process.

20({POSIX_JOB_CONTROL}?C:UNTESTED)

If SIGTSTP is supported:
The default action for SIGTSTP is to stop the process.

21({POSIX_JOB_CONTROL}?C:UNTESTED)

If SIGTTIN is supported:
The default action for SIGTTIN is to stop the process.

22({POSIX_JOB_CONTROL}?C:UNTESTED)

If SIGTTOU is supported:
The default action for SIGTTOU is to stop the process.

D01(C) If the behavior associated with {_POSIX_JOB_CONTROL} is not supported and the support of the signals defined in Table 3-2 of POSIX.1 {3} is documented:
The details are contained in 3.3.1.1 of the PCD.1.

D02(C) if the implementation supports additional signals, and this is documented:
The details are contained in 3.3.1.1 of the PCD.1.

3.3.1.2 Signal Generation and Delivery

23(A) When a signal, whose action is not set to ignore the signal, is blocked and the signal is generated, then the signal remains pending until it is either unblocked or ignored. (See Section B.3.3.1.2.)

24(A) When a signal, whose action is set to ignore the signal, is blocked and the signal is generated, then either it is discarded immediately or it remains pending until either it is unblocked or its action is again set to ignore the signal.

D03(C) If a signal is discarded immediately upon generation or remains pending when the action associated with a blocked signal is SIG_IGN, and this is documented:
The details are confined in 3.3.1.2 of the PCD.1.

25(A) The action taken on delivery of a signal is determined at the time of delivery, taking into account *sigaction*() calls while the signal is pending.

D04(A) Whether a signal is delivered more than once if a subsequent occurrence of a pending signal is generated is described in 3.3.1.2 of the PCD.1.

D05(C) If the order in which multiple simultaneously pending signals are delivered to a process is documented:
The details are contained in 3.3.1.2 of the PCD.1.

26(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:
When any stop signal (SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU) is generated for a process, then any pending SIGCONT signals for that process are discarded.

27(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:
When SIGCONT is generated for a process, then the pending stop signals (SIGTSTP, SIGTTIN, SIGTTOU) for that process are discarded.

28(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:
When SIGCONT is generated for a process that is stopped, then the process continues, even when the SIGCONT signal is blocked or ignored.

29(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:
When SIGCONT is generated for a process that is stopped, and the SIGCONT signal is both blocked and not ignored, then it remains pending until it is either unblocked or a stop signal is generated for the process.

D06(A) When signals are generated under conditions not defined by POSIX.1 {3}, the details are contained in 3.3.1.2 of the PCD.1 or in the subclause of the PCD.1 that corresponds to the subclause in POSIX.1 {3} where the function that provokes the signal is defined.

3.3.1.3 Signal Actions

30(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:
When a process is stopped, then its execution is suspended.

31(A) When the *main*() routine of a new process image is entered, then all signals that are set to the default action or are to be caught in the calling image are set to SIG_DFL in the new process image, and all signals set to be ignored in the calling process image are set to SIG_IGN in the new process image.

32(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:
When a process whose parent has not set the SA_NOCLDSTOP flag stops, a SIGCHLD signal shall be generated for the parent process.

33(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:
When a process is stopped and a SIGABRT, SIGALRM, SIGFPE, SIGHUP, SIGILL, SIGINT, SIGPIPE, SIGQUIT, SIGSEGV, SIGTERM, SIGUSR1, SIGUSR2, or SIGCHLD signal is sent to the process, then the signal is not delivered until the process is continued.

34(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:
When a process is stopped and a SIGKILL is sent to the process, then the signal terminates the receiving process.

35(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:
When a process that is a member of an orphaned process group is sent a SIGTSTP, SIGTTIN, or SIGTTOU signal, then the process is not stopped by this signal, and the signal is discarded.

36(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:
When the action for SIGCHLD is set to SIG_DFL while a SIGCHLD signal is pending, then the pending SIGCHLD signal is discarded.

R01 The system does not allow the action for the signal SIGKILL to be set to SIG_IGN. (See Assertion 20 in 3.3.4.4.)

R02 The system does not allow the action for the signal SIGSTOP to be set to SIG_IGN. (See Assertion 21 in 3.3.4.4.)

37(A) When a signal whose action is set to SIG_IGN is delivered, then the delivery of the signal does not affect the process.

D07(C) If the behavior of a process after it ignores a SIGFPE, SIGILL, or SIGSEGV signal that was not generated by the *kill*() or *raise*() functions is documented:
The details are contained in 3.3.1.3 of the PCD.1.

38(A) When the action for a pending signal that is also blocked is set to SIG_IGN, then the pending signal is discarded.

39(B) When the action for a pending signal that is not blocked is set to SIG_IGN, then the pending signal is discarded.

See Reason 1 in Section 5. of POSIX.3 {4}.

D08(C) If the action taken when a process sets the action for the SIGCHLD signal to SIG_IGN is documented:
The details are contained in 3.3.1.3 of the PCD.1.

40(A) When any one of the signals SIGABRT, SIGALRM, SIGFPE, SIGHUP, SIGILL, SIGINT, SIGPIPE, SIGQUIT, SIGSEGV, SIGTERM, SIGUSR1, or SIGUSR2 is sent to a process whose action is set to catch the signal, then the receiving process executes the signal-catching function at the address specified, and the signal number is passed as the single argument to this function.

41(C) If the behavior associated with {_POSIX_JOB CONTROL} is supported:
When any one of the signals SIGCHLD, SIGCONT, SIGTSTP, SIGTTIN, or SIGTTOU is sent to a process whose action is set to catch the signal, then the receiving process executes the signal-

catching function at the address specified, and the signal number is passed as the signal argument to this function.

42(A) When a process returns from the execution of a signal-catching function, then the process resumes execution from the point at which it was interrupted.

D09(C) If the action taken when a process returns normally from a signal-catching function for a SIGFPE, SIGILL, or SIGSEGV signal that was not generated by the *kill*() or *raise*() function is documented:

The details are contained in 3.3.1.3 of the PCD.1.

43(A) The system does not allow a process to catch the signal SIGKILL.

44(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

The system does not allow a process to catch the signal SIGSTOP.

D10(C) If the action taken when a process establishes a signal-catching function for the SIGCHLD signal while it has a terminated child process for which it has not waited is documented:

Whether a SIGCHLD signal is generated is stated in 3.3.1.3 of the PCD.1.

45(A) The functions *fcntl*(), *kill*(), *open*(), *pause*(), *read*(), *sleep*(), *sigprocmask*(), *sigsuspend*() *wait*(), *waitpid*(), and *write*() are reentrant with respect to signals. That is, they can be called without restriction from a signal-catching function.

46(PCTS_GTI_DEVICE && PCTS_GTI_BUFFERS_OUTPUT?A:UNTESTED)

The function *tcdrain*() is reentrant with respect to signals. That is, *tcdrain*() can be called without restriction from a signal-catching function.

47(B) The functions *_exit*(), *access*(), *alarm*(), *cfgetispeed*(), *cfgetospeed*(), *cfsetispeed*(), *cfsetospeed*(), *chdir*(), *chmod*(), *chown*(), *close*(), *creat*(), *dup*(), *dup2*(), *execle*(), *execve*(), *fork*(), *fstat*(), *getegid*(), *geteuid*(), *getgid*(), *getgroups*(), *getpgrp*(), *getpid*(), *getppid*(), *getuid*(), *link*(), *lseek*(), *mkdir*(), *mkfifo*(), *pathconf*(), *pipe*(), *rename*(), *rmdir*(), *setgid*(), *setpgid*(), *setsid*(), *setuid*(), *sigaction*(), *sigaddset*(), *sigdelset*(), *sigemptyset*(), *sigfillset*(), *sigpending*(), *sigismember*(), *stat*(), *sysconf*(), *tcflow*(), *tcflush*(), *tcgetattr*(), *tcgetpgrp*(), *tcsendbreak*(), *tcsetattr*(), *tcsetpgrp*(), *time*(), *times*(), *umask*(), *uname*(), *unlink*(), and *utime*() are reentrant with respect to signals.

See Reason 3 in Section 5. of POSIX.3 {4}.

48(B) When a function defined by POSIX.1 {3} or by the C Standard {2} is interrupted by a signal that is being caught and a function defined by POSIX.1 {3} or by the C Standard {2} is called from the signal-catching function, then, unless both the interrupted function and the function called from the signal-catching function are marked as unsafe, the behavior is as defined in POSIX.1 {3}.

See Reason 1 in Section 5. of POSIX.3 {4}.

D11(C) If the action taken when a signal interrupts an unsafe function and the signal-catching function calls an unsafe function, and this is documented:

The details on the behavior of those functions is contained in 3.3.1.3 of the PCD.1.

3.3.1.4 Signal Effects on Other Functions

49(A) When a signal is delivered to a process during the execution of any function, and the action associated with the signal is to terminate the process, then the process is terminated, and the original function call does not return.

50(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When a signal is delivered to a process during the execution of any function, and the action associated with the signal is to stop the process, then the process is stopped. A SIGCONT signal sent while the process is stopped causes the original function to continue from the point at which it was stopped.

51(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When a signal is delivered to a process during the execution of any function, and the action associated with the signal is to stop the process, then the process is stopped. A SIGKILL signal sent while the process is stopped causes the process to terminate.

52(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When any of the functions *fcntl*(), *open*(), *pause*(), *read*(), *sleep*(), *sigsuspend*(), *wait*(), *waitpid*(), and *write*() are stopped by the delivery of a signal whose action is to stop the process, then the subsequent generation of a SIGCONT signal that is not being caught by a signal-catching function causes the process to continue without the function reporting that it has been interrupted by either the stopping or the SIGCONT signal.

53(PCTS_GTI_DEVICE && PCTS_GTI_BUFFERS_OUTPUT?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When the function *tcdrain*() is stopped by the delivery of a signal whose action is to stop the process, then the subsequent generation of a SIGCONT signal that is not being caught by a signal-catching function causes the process to continue without the *tcdrain*() function reporting that it has been interrupted by either the stopping or the SIGCONT signal.

54(D) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When any of the functions *_exit*(), *access*(), *alarm*(), *cfgetispeed*(), *cfgetospeed*(), *cfsetispeed*(), *cfsetospeed*(), *chdir*(), *chmod*(), *chown*(), *close*(), *creat*(), *dup*(), *dup2*(), *execle*(), *execve*(), *fork*(), *fstat*(), *getegid*(), *geteuid*(), *getgid*(), *getgroups*(), *getpgrp*(), *getpid*(), *getppid*(), *getuid*(), *kill*(), *link*(), *lseek*(), *mkdir*(), *inkfifo*(), *pathconf*(), *pipe*(), *rename*(), *rmdir*(), *setgid*(), *setpgid*(), *setaid*(), *setuid*(), *sigaction*(), *sigaddset*(), *sigdelset*(), *sigemptyset*(), *sigfillset*(), *sigpending*(), *sigprocmask*(), *sigsimember*(), *stat*(), *sysconf*(), *tcflow*(), *tcflush*(), *tcgetattr*(), *tcgetpgrp*(), *tcsendbreak*(), *tcsetattr*(), *tcsetpgrp*(), *time*(), *times*(), *umask*(), *uname*(), *unlink*(), *ustat*(), and *utime*() are stopped by the delivery of a signal whose action is to stop the process, then the subsequent generation of a SIGCONT signal that is not being caught by a signal-catching function causes the process to continue without the function reporting that it has been interrupted by either the stopping or the SIGCONT signal.

See Reason 3 in Section 5. of POSIX.3 {4}.

55(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When any of the functions *fcntl*(), *open*(), *pause*(), *read*(), *sleep*(), *sigsuspend*(), *wait*(), *waitpid*(), and *write*() are stopped by the delivery of a signal whose action is to stop the process, then the subsequent generation of a SIGCONT signal that is being caught by a signal-catching function causes the process to continue after returning from the signal-catching function, with the original function reporting that it has been interrupted by the SIGCONT signal.

56(PCTS_GTI_DEVICE && PCTS_GTI_BUFFERS_OUTPUT?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When the function *tcdrain*() is stopped by the delivery of a signal whose action is to stop the process, then the subsequent generation of a SIGCONT signal that is being caught by a signal-catching function causes the process to continue after returning from the signal-catching function, with the *tcdrain*() function reporting that it has been interrupted by the SIGCONT signal.

57(D) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When any of the functions *_exit*(), *access*(), *alarm*(), *cfgetispeed*(), *cfgetospeed*(), *cfsetispeed*(), *cfsetospeed*(), *chdir*(), *chmod*(), *chown*(), *close*(), *creat*(), *dup*(), *dup2*(), *execle*(), *execve*(), *fork*(), *fstat*(), *getegid*(), *geteuid*(), *getgid*(), *getgroups*(), *getpgrp*(), *getpid*(), *getppid*(), *getuid*(), *kill*(), *link*(), *lseek*(), *mkdir*(), *mkfifo*(), *pathconf*(), *pipe*(), *rename*(), *rmdir*(), *setgid*(), *setpgid*(), *setsid*(), *setuid*(), *sigaction*(), *sigaddset*(), *sigdelset*(), *sigemptyset*(), *sigfilleet*(), *sigpending*(), *sigprocmask*(), *sigsimember*(), *stat*(), *sysconf*(), *tcflow*(), *tcflush*(), *tcgetattr*(), *tcgetpgrp*(), *tcsendbreak*(), *tcsetattr*(), *tcsetpgrp*(), *time*(), *times*(), *umask*(), *uname*(), *unlink*(), *ustat*(), and *utime*() are stopped

by the delivery of a signal whose action is to stop the process, then the subsequent generation of a SIGCONT signal that is being caught by a signal-catching function causes the process to continue after returning from the signal-catching function, with the original function reporting that it has been interrupted by the SIGCONT signal.

See Reason 3 in Section 5. of POSIX.3 {4}.

58(A) When any signal that is being ignored is generated while any function is being executed, then the signal does not have any effect on the behavior of the function.

Testing Requirements:

Test for the functions *fcntl*(), *open*(), *pause*(), *read*(), *sleep*(), *sigsuspend*(), *wait*(), *waitpid*(), and *write*(), each of which can be placed in a state where they would report being interrupted by a signal if it were delivered. The action of SIGALRM on *sleep*() is unspecified.

59(PCTS_GTI_DEVICE?A:UNTESTED)

When any signal that is being ignored is generated while *tcdrain*() is being executed, then the signal does not have any effect on the behavior of *tcdrain*().

60(A) When any signal that is being blocked is generated while any function is being executed, then the signal does not have any effect on the behavior of the function.

Testing Requirements:

Test for the functions *fcntl*(), *open*(), *pause*(), *read*(), *sleep*(), *sigsuspend*(), *wait*(), *waitpid*(), and *write*(), each of which can be placed in a state where they would report being interrupted by a signal if it were delivered. The action of SIGALRM on *sleep*() is unspecified.

61(PCTS_GTI_DEVICE?A:UNTESTED)

When any signal that is being blocked is generated while *tcdrain*() is being executed, then the signal does not have any effect on the behavior of *tcdrain*().

3.3.2 *kill*()

3.3.2.1 Synopsis

01(A) If the implementation provides C Standard {2} support:

When the header `<signal.h>` is included, then the function prototype `int kill(pid_t, int)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<signal.h>` is included, then the function *kill*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *kill*() is defined as a macro when the header `<signal.h>` is included:

When the macro *kill*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *kill*() is defined as a macro in the header `<signal.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

3.3.2.2 Description

04(A) When a process has the permissions allowing it to send a signal to the process(es) specified by *pid*, then a call to *kill*(*pid*, *sig*) can send any of the signals SIGABRT, SIGALRM, SIGFPE, SIGHUP, SIGILL, SIGINT, SIGKILL, SIGPIPE, SIGQUIT, SIGSEGV, SIGTERM, SIGUSR1, and SIGUSR2. On successful generation of the signal, the return value is zero.

05(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When a process has the permissions allowing it to send a signal to the process(es) specified by *pid*, then a call to *kill*(*pid*, *sig*) can send any of the signals SIGCHLD, SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU, and SIGCONT. On successful generation of the signal, the return value is zero.

06(A) When a process has permission allowing it to send the null signal to the process specified by *pid*, then a call to *kill*(*pid*, *0*) returns a value of zero.

07(A) When *pid* is greater than 0 and the real user ID of the sending process matches the real user ID of the receiving process, then a ca6ll of *kill*(*pid*, *sig*) sends the signal *sig* to the process *pid*.

08(A) When *pid* is greater than 0 and the effective user ID of the sending process matches the real user ID of the receiving process, then a call of *kill*(*pid*, *sig*) sends the signal *sig* to the process *pid*.

09(A) If the behavior associated with {_POSIX_SAVED_IDS} is supported:

When *pid* is greater than 0 and the real user ID of the sending process matches the saved set-user ID of the receiving process, then a call of *kill*(*pid*, *sig*) sends the signal *sig* to the process *pid*.

Otherwise:

When *pid* is greater than 0 and the real user ID of the sending process matches the effective user ID of the receiving process, then a call of *kill*(*pid*, *sig*) sends the signal *sig* to the process *pid*.

10(A) If the behavior associated with {_POSIX_SAVED_IDS} is supported:

When *pid* is greater than 0 and the effective user ID of the sending process matches the saved set-user ID of the receiving process, then a call of *kill*(*pid*, *sig*) sends the signal *sig* to the, process *pid*.

Otherwise:

When *pid* is greater than 0 and the effective user ID of the sending process matches the effective user ID of the receiving process, then a call of *kill*(*pid*, *sig*) sends the signal *sig* to the process *pid*.

D01(C) If the implementation provides a mechanism to associate with a process the appropriate privileges to send signals to processes not associated with the current UID:

The details are documented in 2.2.2.4 or 3.3.2.2 of the PCD.

11(C) If the implementation provides the appropriate privileges to signal any other process:

When a process has the appropriate privileges to send signals to a process *pid* that has different real, effective, or saved set-user IDs, then a call to *kill*(*pid*, *sig*) with *pid* greater than 0 sends the signal *sig* to the process *pid*.

D02(C) If the implementation excludes a set of system processes from being sent a signal *sig* when *pid* is 0, and this is documented:

The set of system processes that are excluded is contained in 3.3.2.2 of the PCD.1.

12(A) A call to *kill*((*pid_t*)0, *sig*) from a process sends the signal *sig* to all processes (excluding the set of system processes defined by the implementation as not receiving the signal) whose process group ID is equal to the process group ID of the sender and for which the process has permission to send the signal.

D03(C) If the behavior of the *kill*() function when *pid* is –1 is documented:

The details are contained in 3.3.2.2 of the PCD.1.

D04(C) If the implementation excludes a set of system processes from being sent a signal *sig* when *pid* is negative, but not -1, and this is documented:

The set of system processes that are excluded is contained in 3.3.2.2 of the PCD.1.

13(A) When *pid* is negative but not –1, a call to *kill*(*pid*, *sig*) causes the signal *sig* to be sent to all process whose process group ID is equal to the absolute value of *pid* and for which the process has permission to send the signal.

14(A) When a call to *kill*(*pid*, *sig*) uses a value of *pid* that causes a signal to be sent to the calling process, and when *sig* is not blocked by the calling process, then either *sig* or at least one pending unblocked signal is delivered to the sending process before the *kill*() function returns.

15(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When the process specified by *pid* is a member of the same session as the calling process, then a call to *kill*(*pid*, *SIGCONT*) sends the SIGCONT signal to the process specified by *pid* irrespective of its user ID.

D05(C) If extended security control imposes restrictions on the sending of signals not defined by POSIX.1 {3}:

The restrictions imposed on the sending of signals, including the NULL signal, are contained in 3.3.2.2 of the PCD.1.

3.3.2.3 Returns

R01 When a call to *kill*() completes successfully, then a value of (*int*)0 is returned. (See Assertions 4–6 in 3.3.2.2.)

R02 When a call to *kill*() completes unsuccessfully, then a value of (*int*)-1 is returned and sets *errno* to indicate the error. (See Assertions 16–20 in 3.3.2.4.)

R03 When a call to *kill*() completes unsuccessfully, no signal is sent. (See Assertions 16–20 in 3.3.2.4.)

3.3.2.4 Errors

16(PCTS_INVALID_signal?A:UNTESTED)

When the value of *sig* is an invalid or an unsupported signal number, then a call to *kill*(*pid*, *sig*) returns a value of (*int*)–1, sets *errno* to [EINVAL], and the signal is not sent.

17(A) If the behavior associated with {_POSIX_SAVED_IDS} is supported:

When the process does not have the appropriate privileges to send signals to the process of another user, and when the real or effective user ID of the sending process does not match the real or saved setuser ID of any receiving process, and when *sig* is not equal to SIGCONT, then a call to *kill*(*pid*, *sig*) returns a value of (*int*)-1, sets *errno* to [EPERM], and the signal is not sent.

Otherwise:

When the sending process does not have the appropriate privileges, and when the real or effective user ID of the sending process does not match the effective user ID of any receiving process, and when *sig* is not equal to SIGCONT, then a call to *kill*(*pid*, *sig*) returns a value of (*int*)-1, sets *errno* to [EPERM], and the signal is not sent.

18(C) If the behaviors associated with {_POSIX_JOB_CONTROL} and {_POSIX_SAVED_IDS} are supported:

When the process does not have the appropriate privileges to send signals to the process of another user, and when the real or effective user ID of the sending process does not match the real or saved setuser ID of any receiving process, and when there are no receiving processes in the same session as the sending process and argument *sig* is equal to SIGCONT, then a call to *kill*(*pid*, *sig*) returns a value of (*int*)-1, sets *errno* to [EPERM], and the SIGCONT signal is not sent.

19(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported and the behavior associated with {_POSIX_SAVED_IDS} is not supported:

When the process does not have the appropriate privileges to send signals to the process of another user, and when the real or effective user ID of the sending process does not match the effective user ID of any receiving process, and when there are no receiving processes in the same session as the sending process and argument *sig* is equal to SIGCONT, then a call to *kill*(*pid*, *sig*) returns a value of (*int*)-1, sets *errno* to [EPERM], and the SIGCONT signal is not sent.

20(A) When no process and no process group can be found corresponding to that specified by *pid*, then a call to *kill*(*pid*, *sig*) returns a value of (*int*)-1, sets *errno* to [ESRCH], and the signal is not sent.

3.3.3 Manipulate Signal Sets

3.3.3.1 *sigemptyset*() (3.3.3)

3.3.3.1.1 Synopsis (3.3.3.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<signal.h>` is included, then the function prototype `int sigemptyset(sigset_t *)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<signal.h>` is included, then the function *sigemptyset*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *sigemptyset*() is defined as a macro when the header `<signal.h>` is included:

When the macro *sigemptyset*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *sigemptyset*() is defined as a macro in the header `<signal.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

3.3.3.1.2 Description (3.3.3.2)

04(A) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

A call to *sigemptyset(set)* initializes the signal set pointed to by the argument set such that the signals SIGABRT, SIGALRM, SIGFPE, SIGHUP, SIGILL, SIGINT, SIGKILL, SIGPIPE, SIGQUIT, SIGSEGV, SIGTERM, SIGUSR1, SIGUSR2, SIGCHLD, SIGCONT, SIGSTOP, SIGTSTP, SIGTTIN, and SIGTTOU are excluded and a value of zero is returned.

Otherwise:

A call to *sigemptyset(set)* initializes the signal set pointed to by the argument set such that the signals SIGABRT, SIGALRM, SIGFPE, SIGHUP, SIGILL, SIGINT, SIGKILL, SIGPIPE, SIGQUIT, SIGSEGV, SIGTERM, SIGUSR1, and SIGUSR2 are excluded and a value of zero is returned.

D01(C) If the action taken when an object of type *sigset_t* has not been initialized by a call to *sigemptyset*() or *sigfillset*(), but a pointer to the object is supplied as an argument to the functions *sigaddset*(), *sigdelset*(), *sigismember*(), *sigaction*(), *sigrocmask*(), *sigpending*(), or *sigsuspend*(), and this is documented:

The details are contained in 3.3.3.2 of the PCD.1.

3.3.3.1.3 Returns (3.3.3.3)

R01 When a call to *sigemptyset*() completes successfully, then a value of (*int*)0 is returned. (See Assertion 4 in 3.3.3.1.2.)

3.3.3.1.4 Errors (3.3.3.4)

There are no assertions specific to this subclause.

3.3.3.2 *sigfillset*() (3.3.3)

3.3.3.2.1 Synopsis (3.3.3.1)

01(A) If the implementation provides C Standard {2} support:

When the header <signal.h> is included, then the function prototype `int sigfillset(sigset_t *)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header <signal.h> is included, then the function *sigfillset*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *sigfillset*() is defined as a macro when the header <signal.h> is included:

When the macro *sigfillset*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *sigfillset*() is defined as a macro in the header <signal.h>:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

3.3.3.2.2 Description (3.3.3.2)

04(A) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

A call to *sigfillset*(*set*) initializes the signal set pointed to by the argument *set* such that the signals SIGABRT, SIGALRM, SIGFPE, SIGHUP, SIGILL, SIGINT, SIGKILL, SIGPIPE, SIGQUIT, SIGSEGV, SIGTERM, SIGUSR1, SIGUSR2, SIGCHLD, SIGCONT, SIGSTOP, SIGTSTP, SIGTTIN, and SIGTTOU are included and a value of zero is returned.

Otherwise:

A call to *sigfillset*(*set*) initializes the signal set pointed to by the argument *set* such that the signals SIGABRT, SIGALRM, SIGFPE, SIGHUP, SIGILL, SIGINT, SIGKILL, SIGPIPE, SIGQUIT, SIGSEGV, SIGTERM, SIGUSR1, and SIGUSR2 are included and a value of zero is returned.

3.3.3.2.3 Returns (3.3.3.3)

R01 When a call to *sigfillset*() completes successfully, then a value of (*int*)0 is returned. (See Assertion 4 in 3.3.3.2.2.)

3.3.3.2.4 Errors (3.3.3.4)

There are no assertions specific to this subclause.

3.3.3.3 *sigaddset*() (3.3.3)

3.3.3.3.1 Synopsis (3.3.3.1)

01(A) If the implementation provides C Standard {2} support:

When the header <signal.h> is included, then the function prototype `int sigaddset(sigset_t *, int)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header <signal.h> is included, then the function *sigaddset*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *sigaddset*() is defined as a macro when the header <signal.h> is included:

When the macro *sigaddset*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *sigaddset*() is defined as a macro in the header <signal.h>:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

3.3.3.3.2 Description (3.3.3.2)

04(A) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

A call to *sigaddset*(*set*, *signo*) where the value of the *signo* argument is SIGABRT, SIGALRM, SIGFPE, SIGHUP, SIGILL, SIGINT, SIGKILL, SIGPIPE, SIGQUIT, SIGSEGV, SIGTERM, SIGUSR1, SIGUSR2, SIGCHLD, SIGCONT, SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU adds the *signo* argument to the signal set pointed to by the argument *set*, and a value of zero is returned.

Otherwise:

A call to *sigaddset*(*set*, *signo*) where the value of the *signo* argument is SIGABRT, SIGALRM, SIGFPE, SIGHUP, SIGILL, SIGINT, SIGKILL, SIGPIPE, SIGQUIT, SIGSEGV, SIGTERM, SIGUSR1, or SIGUSR2 adds the *signo* argument to the signal set pointed to by the argument *set*, and a value of zero is returned.

3.3.3.3.3 Returns (3.3.3.3)

R01 When a call to *sigaddset*() completes successfully, then a value of (*int*)0 is returned. (See Assertion 4 in 3.3.3.3.2.)

R02 When a call to *sigaddset*() completes unsuccessfully, then a value of (*int*)–1 is returned and sets *errno* to indicate the error. (See Assertion 5 in 3.3.3.3.4.)

3.3.3.3.4 Errors (3.3.3.4)

D01(C) If the implementation supports the detection of [EINVAL] for *sigaddset*():

The details under which [EINVAL] occurs for *sigaddset*() are contained in 3.3.3.4 of the PCD.1. (See DGA02 in 2.4.)

05(PCTS_INVALID_SIGNAL?C:UNTESTED)

If the implementation supports the detection of [EINVAL] for *sigaddset*():

When the value of the argument *signo* is an invalid or unsupported signal number, then a call to *sigaddset*(*set*, *signo*) returns a value of (*int*)-1 and sets *errno* to [EINVAL]. The signal set pointed to by the argument *set* is not changed.

06(D) If the implementation does not support the detection of [EINVAL] for *sigaddset*():

A call to *sigaddset*(*set*, *signo*) is successful for all possible values of *signo* (unless a different error condition is detected). (See GA26 in 2.4.)

See Reason 2 in Section 5. of POSIX.3 {4}.

3.3.3.4 *sigdelset*() (3.3.3)

3.3.3.4.1 Synopsis (3.3.3.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<signal.h>` is included, then the function prototype
`int sigdelset(sigset_t *, int)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<signal.h>` is included, then the function *sigdelset*() is either declared with result type int or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *sigdelset*() is defined as a macro when the header `<signal.h>` is included:

When the macro *sigdelset*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *sigdelset*() is defined as a macro in the header `<signal.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

3.3.3.4.2 Description (3.3.3.2)

04(A) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

A call to *sigdelset*(*set, signo*) where the value of the *signo* argument is SIGABRT, SIGALRM, SIGFPE, SIGHUP, SIGILL, SIGINT, SIGKILL, SIGPIPE, SIGQUIT, SIGSEGV, SIGTERM, SIGUSR1, SIGUSR2, SIGCHLD, SIGCONT, SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU deletes the *signo* argument from the signal set pointed to by the argument *set*, and a value of zero is returned.

Otherwise:

A call to *sigdelset*(*set, signo*) where the value of the *signo* argument is SIGABRT, SIGALRM, SIGFPE, SIGHUP, SIGILL, SIGINT, SIGKILL, SIGPIPE, SIGQUIT, SIGSEGV, SIGTERM, SIGUSR1, or SIGUSR2 deletes the *signo* argument from the signal set pointed to by the argument *set*, and a value of zero is returned.

3.3.3.4.3 Returns (3.3.3.3)

R01 When a call to *sigdelset*() completes successfully, then a value of (*int*)0 is returned. (See Assertion 4 in 3.3.3.4.2.)

R02 When a call to *sigdelset*() completes unsuccessfully, then a value of (*int*)-1 is returned and sets *errno* to indicate the error. (See Assertion 5 in 3.3.3.4.4.)

3.3.3.4.4 Errors (3.3.3.4)

D01(C) If the implementation supports the detection of [EINVAL] for *sigdelset*():

The details under which [EINVAL] occurs for *sigdelset*() are contained in 3.3.3.4 of the PCD.1. (See DGA02 in 2.4.)

05(PCTS_INVALID_SIGNAL?C:UNTESTED)

If the implementation supports the detection of [EINVAL] for *sigdelset*():

When the value of the argument *signo* is an invalid or unsupported signal number, then a call to *sigdelset*(*set, signo*) returns a value of (*int*)-1 and sets *errno* to [EINVAL]. The signal set pointed to by the argument *set* is not changed.

06(D) If the implementation does not support the detection of [EINVAL] for *sigdelset*():

A call to *sigdelset*(*set, signo*) is successful for all possible values of *signo* (unless a different error condition is detected). (See GA26 in 2.4.)

See Reason 2 in Section 5. of POSIX.3 {4}.

3.3.3.5 *sigismember*() (3.3.3)

3.3.3.5.1 Synopsis (3.3.3.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<signal.h>` is included, then the function prototype `int sigismember(const sigset_t *, int)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header <signal.h> is included, then the function *sigismember*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *sigismember*() is defined as a macro when the header <signal.h> is included:

When the macro *sigismember*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *sigismember*() is defined as a macro in the header <signal.h>:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

3.3.3.5.2 Description (3.3.3.2)

04(A) If the behavior associated with {_POSIX JOB_CONTROL} is supported:

A call to *sigismember*(*set, signo*) where the value of the *signo* argument is SIGABRT, SIGALRM, SIGFPE, SIGHUP, SIGILL, SIGINT, SIGKILL, SIGPIPE, SIGQUIT, SIGSEGV, SIGTERM, SIGUSR1, SIGUSR2, SIGCHLD, SIGCONT, SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU tests whether the *signo* argument is a member of the signal set pointed to by the argument *set* and returns a value of 1 if the specified signal is a member of the set and a value of 0 if it is not.

Otherwise:

A call to *sigismember*(*set, signo*) where the value of the *signo* argument is SIGABRT, SIGALRM, SIGFPE, SIGHUP, SIGILL, SIGINT, SIGKILL, SIGPIPE, SIGQUIT, SIGSEGV, SIGTERM, SIGUSR1, or SIGUSR2 tests whether the *signo* argument is a member or the signal set pointed to by the argument *set* and returns a value of 1 if the specified signal is a member of the set and a value of 0 if it is not.

3.3.3.5.3 Returns (3.3.3.3)

R01 When the value of the *signo* argument is a member of *set*, then a call to *sigismember*(*set, signo*) returns a value of (*int*)1. (See Assertion 4 in 3.3.3.5.2.)

R02 When the value of the *signo* argument is not a member of *set*, then a call to *sigismember*(*set, signo*) returns a value of (*int*)0. (See Assertion 4 in 3.3.3.5.2.)

R03 When a call to *sigismember*() completes unsuccessfully, then a value of (*int*)–1 is returned and sets *errno* to indicate the error. (See Assertion 5 in 3.3.3.5.4.)

3.3.3.5.4 Errors (3.3.3.4)

D01(C) If the implementation supports the detection of [EINVAL] for *sigismember*():

The details under which [EINVAL] occurs for *sigismember*() are contained in 3.3.3.4 of the PCD.1. (See DGA02 in 2.4.)

05(PCTS_INVALID_SIGNAL?C:UNTESTED)

If the implementation supports the detection of [EINVAL] for *sigismember*():

When the value of the argument *signo* is an invalid or unsupported signal number, then a call to *sigismember*(*set, signo*) returns a value of (*int*)-1 and sets *errno* to [EINVAL].

06(D) If the implementation does not support the detection of [EINVAL] for *sigismember*():

A call to *sigismember*(*set, signo*) is successful for all possible values of *signo* (unless a different error condition is detected). (See GA26 in 2.4.)

See Reason 2 in Section 5. of POSIX.3 {4}.

3.3.4 *sigaction* ()

3.3.4.1 Synopsis

01(A) If the implementation provides C Standard {2} support:

When the header `<signal.h>` is included, then the function prototype `int sigaction(int, const struct sigaction *, struct sigaction *)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<signal.h>` is included, then the function *sigaction*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *sigaction*() is defined as a macro when the header `<signal.h>` is included:

When the macro *sigaction*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *sigaction*() is defined as a macro in the header `<signal.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

3.3.4.2 Description

04(A) When the header `<signal.h>` is included, then the structure *sigaction* is defined, and this structure defines the following members, as shown in Table 3.1.

Table 3.1—Structure *sigaction* Members

Member	Type
sa_handler	*void(*)*()
sa_mask	*sigset_t*
sa_flags	*int*

05(A) A call to *sigaction*(*sig, NULL, oact*) does not alter the signal handling, places the action currently associated with the signal in the location pointed to by the non-NULL argument *oact*, and returns a value of zero.

Testing Requirements:

When the *sigaction* structure contains additional members beyond those specified in POSIX.1 {3} and the effect of these members is not activated, then these additional structure members do not affect the behavior of the call to *sigaction*().

06(A) A call to *sigaction*(*sig, act, NULL*) associates the action specified by a non-NULL *act* with the signal *sig* and returns a value of zero.

Testing Requirements:

When the *sigaction* structure contains additional members beyond those specified in POSIX.1 {3} and the effect of these members is not activated, then these additional structure members do not effect the behavior of the call to *sigaction*().

07(A) A call to *sigaction*(*sig, act, oact*) associates the action specified by a non-NULL *act* with the signal *sig* and stores the action previously associated with the signal in the location pointed to by the non-NULL argument *oact*.

Testing Requirements:

When the *sigaction* structure contains additional members beyond those specified in POSIX.1 {3} and the effect of these members is not activated, then these additional structure members do not effect the behavior of the call to *sigaction*().

08(A) The *sa_handler* field of the *sigaction* structure can be set to SIG_DFL, SIG_IGN, or to a signal-handling function.

Testing Requirements:

When the *sigaction* structure contains additional members beyond those specified in POSIX.1 {3} and the effect of these members is not activated, then these additional structure members do not effect the behavior of the call to *sigaction*().

09(A) When a signal is delivered, then the action defined in the *sa_handler* field of the *sigaction* structure associated with the signal at the time of delivery is performed.

Testing Requirements:

When the *sigaction* structure contains additional members beyond those specified in POSIX.1 {3}; and the effect of these members is not activated, then these additional structure members do not effect the behavior of the call to *sigaction*().

10(A) When a signal is caught by a signal-catching function installed by *sigaction*(), then a new signal mask is calculated by taking the union of the current signal mask, the *sigaction* signal mask for the signal being delivered, and the signal being delivered. The signals SIGSTOP and SIGKILL are not included in the new signal mask. The new signal mask is installed before entering the signal-catching function and remains in effect until

- The signal catching function returns,
- The function *sigprocmask*() is called, or
- The function *sigsuspend*() is called.

Testing Requirements:

When the *sigaction* structure contains additional members beyond those specified in POSIX.1 {3} and the effect of these members is not activated, then these additional structure members do not effect the behavior of the call to *sigaction*().

11(A) When the signal handler for the user returns normally, then the original signal mask is restored.

Testing Requirements:

When the *sigaction* structure contains additional members beyond those specified in POSIX.1 {3} and the effect of these members is not activated, then these additional structure members do not effect the behavior of the call to *sigaction*().

12(A) When the header `<signal.h>` is included, then the flag SA_NOCLDSTOP is defined and can be set in *sa_flags*.

Testing Requirements:

When the *sigaction* structure contains additional members beyond those specified in POSIX.1 {3} and the effect of these members is not activated, then these additional structure members do not effect the behavior of the call to *sigaction*().

13(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When *sig* is set to SIGCHLD and the SA_NOCLDSTOP flag is clear in *sa_flags*, then a call to *sigaction*(*sig, act, oact*) causes a SIGCHLD signal to be generated for the parent process whenever any of its child processes stop.

Testing Requirements:

When the *sigaction* structure contains additional members beyond those specified in POSIX.1 {3} and the effect of these members is not activated, then these additional structure members do not effect the behavior of the call to *sigaction*().

14(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When *sig* is SIGCHLD and the SA_NOCLDSTOP flag is set in *sa_flags*, then a call to *sigaction*(*sig, act, oact*) results in a SIGCHLD signal not being generated for the parent process whenever any of its child processes stop.

Testing Requirements:

When the *sigaction* structure contains additional members beyond those specified in POSIX.1 {3} and the effect of these members is not activated, then these additional structure members do not effect the behavior of the call to *sigaction*().

15(A) When an action is installed for a specific signal, then it remains installed until another action is explicitly requested by another call to *sigaction*() or until one of the `exec`() functions is called.

Testing Requirements:

Testing the change of signal action associated with a call to `exec`() need not be tested as part of this assertion; it is more thoroughly covered in other places.

When the *sigaction* structure contains additional members beyond those specified in POSIX.1 {3} and the effect of these members is not activated, then these additional structure members do not effect the behavior of the call to *sigaction*().

D01(C) If the action taken when *sigaction*() is called, when the previous action for signal *sig* had been established by the *signal*() function as defined in the C Standard {2}, is documented:

The details on the values of the fields returned in the structure pointed to by *oact* for a call to *sigaction*() are contained in 3.3.4.2 of the PCD.1.

16(C) If the function *signal*() is supported:

When a signal-handling function has been installed using the *signal*() function, then the value supplied in the *oact* structure member *sa_handler* on a call to *sigaction*() can be used by *sigaction*() to reestablish the original signal-handling function.

Testing Requirements:

When the *sigaction* structure contains additional members beyond those specified in POSIX.1 {3}; and the effect of these members is not activated, then these additional structure members do not effect the behavior of the call to *sigaction*().

R01 When *sigaction*() fails, then no new signal handler is installed. (See Assertions 20 and 21 in 3.3.4.4.)

D02(C) If the action taken when an attempt to set the action for a signal that cannot be caught or ignored to SIG_DFL is made, and this is documented:

The details on whether the action is ignored or causes an error to be returned are contained in 3.3.4.2 of the PCD.1.

17(A) When *sigaction*(*sig*, *act*, *oact*) is called to set the action for SIGKILL to SIG_DFL, then either a value of -1 is returned and *errno* is set to [EINVAL] or a value of 0 is returned.

Testing Requirements:

When the *sigaction* structure contains additional members beyond those specified in POSIX.1 {3} and the effect of these members is not activated, then these additional structure members do not effect the behavior of the call to *sigaction*().

18(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When *sigaction*(*sig*, *act*, *oact*) is called to set the action for SIGSTOP to SIG_DFL, then either a value of -1 is returned and *errno* is set to [EINVAL] or a value of 0 is returned.

Testing Requirements:

When the *sigaction* structure contains additional members beyond those specified in POSIX.1 {3} and the effect of these members is not activated, then these additional structure members do not effect the behavior of the call to *sigaction*().

3.3.4.3 Returns

R02 When a call to *sigaction*() completes successfully, then a value of (*int*)0 is returned. (See Assertions 5 and 6 in 3.3.4.2.)

R03 When a call to *sigaction*() completes unsuccessfully, then a value of (*int*)-1 is returned and sets *errno* to indicate the error. (See Assertions 19-21 in 3.3.4.4.)

3.3.4.4 Errors

19(PCTS_INVALID_SIGNAL?A:UNTESTED)

When the value of the *sig* argument is an invalid or unsupported signal number, then a call to *sigaction*(*sig*, *act*, *oact*) returns a value of (*int*)-1 and sets *errno* to [EINVAL].

20(A) A call to *sigaction*(*sig*, *act*, *oact*) to catch or ignore SIGKILL returns a value of (*int*)-1, sets *errno* to [EINVAL], and does not install a new handler.

21(C) If the behavior associated with SIGSTOP is supported:

A call to *sigaction*(*sig*, *act*, *oact*) to catch or ignore SIGSTOP returns a value of (*int*)-1, sets *errno* to [EINVAL], and no new handler is installed.

R04 An attempt to set the action to SIG_DFL for signal SIGKILL or SIGSTOP either returns a value of [(*int*)-1] and sets *errno* to [EINVAL], or returns a value of [(*int*)0]. (See Assertions 17 and 18 in 3.3.4.2.)

3.3.5 *sigprocmask*()

3.3.5.1 Synopsis

01(A) If the implementation provides C Standard [2] support:

When the header `<signal.h>` is included, then the function prototype `int sigprocmask(int,const sigset_t *, sigset_t *)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<signal.h>` is included, then the function *sigprocmask*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *sigprocmask*() is defined as a macro when the header `<signal.h>` is included:

When the macro *sigprocmask*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *sigprocmask*() is defined as a macro in the header `<signal.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

3.3.5.2 Description

04(A) When the header `<signal.h>` is included, then the symbols SIG_BLOCK, SIG_UNBLOCK, and SIG_SETMASK are defined.

05(A) A call to *sigprocmask*(*SIG_BLOCK*, *set*, *oset*) changes the signals blocked for the process to the union of the current blocked signals and the signal set pointed to by the non-NULL argument *set*, and returns zero. When *oset* is non-NULL, the previous mask is stored in the location pointed to by *oset*.

06(A) A call to *sigprocmask*(*SIG_UNBLOCK*, *set*, *oset*) changes the signals blocked for the process to the intersection of the current blocked signals and the complement of the signal set pointed to by the non-NULL argument *set*, and returns zero. When *oset* is non-NULL, the previous mask is stored in the location pointed to by *oset*.

07(A) A call to *sigprocmask*(*SIG_SETMASK*, *set*, *oset*) changes the signals blocked for the process to the signal set pointed to by the non-NULL argument *set* and returns zero. When *oset* is non-NULL, the previous mask is stored in the location pointed to by *oset*.

08(A) A call to *sigprocmask*(*how*, *NULL*, *oset*) does not change the signal mask of the process. When *oset* is non-NULL, the previous mask is stored in the location pointed to by *oset*.

09(A) A call to *sigprocmask*(*how*, *set*, *NULL*) changes the signals blocked for the process according to the directive *how* and the signal set pointed to by the non-NULL argument *set*.

10(A) A call to *sigprocmask*(*how*, *NULL*, *NULL*) does not change the signal mask of the process.

11(A) When SIGKILL is included in the set pointed to by the set argument, then the *sigprocmask*() call succeeds, and SIGKILL is not blocked from delivery.

12(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:
When SIGSTOP is included in the set argument, then the *sigprocmask*() call succeeds, and SIGSTOP is not blocked from delivery.

D01(C) If the action taken when any of the signals SIGFPE, SIGILL, or SIGSEGV are generated while blocked, and the signal was not generated by a call to *kill*() or *raise*(), and this is documented:
The details on the action to be taken by the implementation are contained in 3.3.5.2 of the PCD.1.

R01 When the *sigprocmask*() function fails, then the signal mask of the process is not changed. (See Assertion 14 in 3.3.5.4.)

13(A) When there are any pending unblocked signals after the call to the *sigprocmask*() function, then at least one of those signals is delivered before the *sigprocmask*() function returns.

3.3.5.3 Returns

R02 When a call to *sigprocmask*() completes successfully, then a value of (*int*)0 is returned. (See Assertions 5-7 in 3.3.5.2.)

R03 When a call to *sigprocmask*() completes unsuccessfully, then a value of (*int*)-1 is returned and sets *errno* to indicate the error. (See Assertion 14 in 3.3.5.4.)

3.3.5.4 Errors

14(A) When the value of the *how* argument is not one of the defined values and *set* is non-NULL, then a call to *sigprocmask*(*how*, *set*, *oset*) returns a value of (*int*)−1, sets *errno* to [EINVAL], and does not change the signal mask of the process.

3.3.6 *sigpending*()

3.3.6.1 Synopsis

01(A) If the implementation provides C Standard {2} support:
When the header `<signal.h>` is included, then the function prototype
`int sigpending(sigset_t *)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<signal.h>` is included, then the function *sigpending*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

002(C) If *sigpending*() is defined as a macro when the header `<signal.h>` is included:
When the macro *sigpending*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *sigpending*() is defined as a macro in the header `<signal.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

3.3.6.2 Description

04(A) A call to *sigpending*(*set*) stores the set of signals that are pending and blocked from delivery in the calling process in the space pointed to by the argument set, and returns a value of zero.

3.3.6.3 Returns

R01 When a call to *sigpending*() completes successfully, then a value of (*int*)0 is returned. (See Assertion 4 in 3.3.6.2.)

05(D) If the implementation supports error conditions for *sigpending*():

When a call to *sigpending*() completes unsuccessfully, then a value of (*int*)-1 is returned and sets *errno* to indicate the error.

See Reason 2 in Section 5. of POSIX.3 {4}.

3.3.6.4 Errors

D01(C) If the implementation supports error conditions for *sigpending*(), and this is documented:

The conditions under which errors are detected for *sigpending*() is stated in 3.3.6.4 of the PCD.1. (See DGA02 in 2.4.)

3.3.7 *sigsuspend*()

3.3.7.1 Synopsis

01(A) If the implementation provides C Standard {2} support:

When the header `<signal.h>` is included, then the function prototype `int sigsuspend(const sigset_t *)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<signal.h>` is included, then the function *sigsuspend*() is either declared with result type int or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *sigsuspend*() is defined as a macro when the header `<signal.h>` is included:

When the macro *sigsuspend*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *sigsuspend*() is defined as a macro in the header `<signal.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

3.3.7.2 Description

04(A) A call to *sigsuspend*(*sigmask*) replaces the signal mask of the calling process with the set of signals pointed to by the argument sigmask and then suspends the process until delivery of a signal whose action is to execute a signal-catching function or to terminate the process.

05(A) When the action of the signal delivered to the suspended by a call to *sigsuspend*() is to terminate the process, then the *sigsuspend*() function does not return.

06(A) When the action of the signal delivered to the process suspended by a call to *sigsuspend*() is to execute a signal-catching function, then on return *sigsuspend*() restores the signal mask to the set that existed prior to the *sigsuspend*() call.

07(A) A call to *sigsuspend*() attempting to block SIGKILL will cause no error, and the receipt of the SIGKILL will result in the default action for that signal.

08(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:
A call to *sigsuspend*() attempting to block SIGSTOP will cause no error, and the receipt of SIGSTOP will result in the default action for that signal.

3.3.7.3 Returns

R01 A call to *sigsuspend*() is always unsuccessful. It returns a value of [(*int*)-1] and sets *errno* to indicate the error. (See Assertion 9 in 3.3.7.4.)

3.3.7.4 Errors

09(A) When a signal is caught by the calling process and control is returned from the signal catching function, then a call to *sigsuspend*() returns a value of (*int*)-1 and sets *errno* to [EINTR].

3.4 Timer Operations

3.4.1 *alarm*()

3.4.1.1 Synopsis

01(A) If the implementation provides C Standard {2} support:
When the header `<unistd.h>` is included, then the function prototype `unsigned int alarm(unsigned int)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<unistd.h>` is included, then the function *alarm*() is declared with the result type *unsigned int*. (See GA36 in 2.7.3.)

02(C) If *alarm*() is defined as a macro when the header `<unistd.h>` is included:
When the macro *alarm*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *unsigned int*. (See GA37 in 2.7.3.)

03(C) If *alarm*() is defined as a macro in the header `<unistd.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

3.4.1.2 Description

04(A) When the value of the argument seconds is nonzero, then a call to *alarm*(*seconds*) causes the system to send the calling process a SIGALRM signal after the number of real-time seconds specified by the argument *seconds* have elapsed.

05(A) Successive calls to *alarm*() reschedule the alarm clock for the calling process.

06(A) A call to *alarm*(0) cancels any previously made *alarm*() request.

3.4.1.3 Returns

07(A) When there is a previous *alarm*() request with at least 1 s remaining, the return value of *alarm*() is the amount of time remaining in seconds before the system is scheduled to generate the SIGALRM signal.

08(A) When there is no previous *alarm*() request, the return value of *alarm*() is zero.

09(A) When a call to *alarm*() returns and there is a previous *alarm*() request with less than 1 s remaining, then the return value is 1.

3.4.1.4 Errors

There are no assertions specific to this subclause.

3.4.2 *pause*()

3.4.2.1 Synopsis

01(A) If the implementation provides C Standard {2} support:
When the header `<unistd.h>` is included, then the function prototype `inc pause(void)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<unistd.h>` is included, then the function *pause*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *pause*() is defined as a macro when the header `<unistd.h>` is included:
When the macro *pause*() is invoked, then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *pause*() is defined as a macro in the header `<unistd.h>`:
It protects its return value with extra parentheses when necessary. (See GA01 in 1.3.4.)

3.4.2.2 Description

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

05(A) When a signal is delivered and the action is to terminate the process, then *pause*() does not return.

06(A) When a signal is delivered and the signal action is to execute a signal-catching function, then *pause*() returns after the signal-catching function returns.

3.4.2.3 Returns

R01 A call to *pause*() is always unsuccessful. It returns a value of (*int*)-1 and sets *errno* to [EINTR]. (See Assertion 7 in 3.4.2.4.)

3.4.2.4 Errors

07(A) When a signal is caught by the calling process and control is returned from the signal-catching function, then a call to *pause*() returns a value of (*int*)-1 and sets *errno* to [EINTR].

3.4.3 *sleep*()

3.4.3.1 Synopsis

01(A) If the implementation provides C Standard {2} support:
When the header `<unistd.h>` is included, then the function prototype `unsigned int sleep(unsigned int)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<unistd.h>` is included, then the function *sleep*() is declared with the result type *unsigned int*. (See GA36 in 2.7.3.)

02(C) If *sleep*() is defined as a macro when the header `<unistd.h>` is included:

When the macro *sleep*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *unsigned int*. (See GA37 in 2.7.3.)

03(C) If *sleep*() is defined as a macro in the header `<unistd.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

3.4.3.2 Description

04(A) When the number of real-time seconds specified by the value of the argument *seconds* have elapsed since a call of *sleep*(*seconds*) was made, then the call to *sleep*(*seconds*) returns a value of zero.

05(A) When a signal is delivered before the number of real-time seconds specified by the value of the argument *seconds* have elapsed, and when the action associated with the signal is to invoke a signal-catching function, and when this signal-catching function does not alter the disposition or the action associated with the SIGALRM function, then on return from the signal-catching function the call to *sleep*(*seconds*) returns the unslept time in seconds.

06(A) When a signal is delivered before the number of real-time seconds specified by the value of the argument *seconds* have elapsed, and when the action associated with this signal is to terminate the process, then the call to *sleep*(*seconds*) does not return.

D01(C) If the action taken, when the SIGALRM signal generated for the calling process during execution of the *sleep*() function is being ignored or blocked from delivery, is documented:

The details on whether *sleep*() returns when the SIGALRM signal is scheduled are contained in 3.4.3.2 of the PCD.1.

D02(C) If the action taken, when the SIGALRM signal is blocked from delivery, is documented:

The details on whether SIGALRM is discarded or remains pending after the *sleep*() function returns are contained in 3.4.3.2 of the PCD.1.

D03(C) If the action taken, when the SIGALRM signal generated for the calling process during execution of the *sleep*() function—except as a result of a prior call to *alarm*()—is not being ignored or blocked from delivery, is documented:

The details on whether signal SIGALRM has any effect other than causing the *sleep*() function to return is contained in 3.4.3.2 of the PCD.1.

D04(C) If the action taken with the SIGALRM signal—whether the SIGALRM signal is blocked from delivery—when a signal-catching function interrupts the *sleep*() function and either examines or changes the time a SIGALRM is scheduled to be delivered, is documented:

The time the SIGALRM signal is to be delivered, the action associated with the SIGALRM signal, and whether the SIGALRM signal is blocked from delivery are contained in 3.4.3.2 of the PCD.1.

D05(C) If the action taken, when a signal-catching function interrupts *sleep*() and calls *siglongjmp*() or *longjmp*() to restore an environment saved prior to the *sleep*() call, is documented:

The details on the action associated with the SIGALRM signal, the time at which a SIGALRM signal is scheduled to be generated, and the action associated when the SIGALRM signal is blocked and the signal mask of the process is not restored as part of the environment are contained in 3.4.3.2 of the PCD.1.

3.4.3.3 Returns

R01 When a call to *sleep*() returns because the requested time has elapsed, a value of zero is returned. (See Assertion 4 in 3.4.3.2.)

R02 When interrupted by an alarm or another signal, then a call to *sleep*() returns the unslept time in seconds. (See Assertion 5 in 3.4.3.2.)

3.4.3.4 Errors

There are no assertions specific to this subclause.

4. Process Environment

4.1 Process Identification

4.1.1 Get Process and Parent Process IDs

4.1.1.1 *getpid*() (4.1.1)

4.1.1.1.1 Synopsis (4.1.1.1)

01(A) If the implementation provides C Standard {2} support:
When the header `<unistd.h>` is included, then the function prototype `pid_t getpid(void)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<unistd.h>` is included, then the function *getpid*() is declared with the result type *pid_t*. (See GA36 in 2.7.3.)

02(C) If *getpid*() is defined as a macro when the header `<unistd.h>` is included:
When the macro *getpid*() is invoked, then it expands to an expression with the result type *pid_t*. (See GA37 in 2.7.3.)

03(C) If *getpid*() is defined as a macro in the header `<unistd.h>`:
It protects its return value with extra parentheses when necessary. (See GA01 in 1.3.4.)

4.1.1.1.2 Description (4.1.1.2)

04(A) A call to *getpid*() returns the process ID of the calling process.

4.1.1.1.3 Returns (4.1.1.3)

There are no assertions specific to this subclause.

4.1.1.1.4 Errors (4.1.1.4)

There are no assertions specific to this subclause.

4.1.1.2 *getppid*() (4.1.1)

4.1.1.2.1 Synopsis (4.1.1.1)

01(A) If the implementation provides C Standard {2} support:
When the header `<unistd.h>` is included, then the function prototype `pid_t getppid(void)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<unistd.h>` is included, then the function *getppid*() is declared with the result type *pid_t*. (See GA36 in 2.7.3.)

02(C) If *getppid*() is defined as a macro when the header `<unistd.h>` is included:

When the macro *getppid*() is invoked, then it expands to an expression with the result type *pid_t*. (See GA37 in 2.7.3.)

03(C) If *getppid*() is defined as a macro in the header `<unistd.h>`:
It protects its return value with extra parentheses when necessary. (See GA01 in 1.3.4.)

4.1.1.2.2 Description (4.1.1.2)

04(A) A call to *getppid()* returns the parent process ID of the calling process.

4.1.1.2.3 Returns (4.1.1.3)

There are no assertions specific to this subclause.

4.1.1.2.4 Errors (4.1.1.4)

There are no assertions specific to this subclause.

4.2 User Identification

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

4.2.1.1 *getuid*() (4.2.1)

4.2.1.1.1 Synopsis (4.2.1.1)

01(A) If the implementation provides C Standard {2} support:
When the header `<unistd.h>` is included, then the function prototype `uid_t getuid(void)` is declared. (see GA36 in 2.7.3.)

Otherwise:
When the header `<unistd.h>` is included, then the function *getuid*() is declared with the result type *uid_t*. (see GA36 in 2.7.3.)

02(C) If *getuid*() is defined as a macro when the header `<unistd.h>` is included:
When the macro *getuid*() is invoked, then it expands to an expression with the result type *uid_t*. (see GA37 in 2.7.3.)

03(C) If *getuid*() is defined as a macro in the header `<unistd.h>`:
It protects its return value with extra parentheses when necessary. (See GA01 in 1.3.4.)

4.2.1.1.2 Description (4.2.1.2)

04(A) A call to *getuid*() returns the real user ID of the calling process.

4.2.1.1.3 Returns (4.2.1.3)

There are no assertions specific to this subclause.

4.2.1.1.4 Errors (4.2.1.4)

There are no assertions specific to this subclause.

4.2.1.2 *geteuid*() (4.2.1)

4.2.1.2.1 Synopsis (4.2.1.1)

01(A) If the implementation provides C Standard {2} support:
When the header `<unistd.h>` is included, then the function prototype `uid_t geteuid(void)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<unistd.h>` is included, then the function *geteuid*() is declared with the result type *uid_t*. (See GA36 in 2.7.3.)

02(C) If *geteuid*() is defined as a macro when the header `<unistd.h>` is included:
When the macro *geteuid*() is invoked, then it expands to an expression with the result type *uid_t*. (See GA37 in 2.7.3.)

03(C) If *geteuid*() is defined as a macro in the header `<unistd.h>`:
It protects its return value with extra parentheses when necessary. (See GA01 in 1.3.4.)

4.2.1.2.2 Description (4.2.1.2)

04(A) A call to *geteuid*() returns the effective user ID of the calling process.

4.2.1.2.3 Returns (4.2.1.3)

There are no assertions specific to this subclause.

4.2.1.2.4 Errors (4.2.1.4)

There are no assertions specific to this subclause.

4.2.1.3 *getgid*() (4.2.1)

4.2.1.3.1 Synopsis (4.2.1.1)

01(A) If the implementation provides C Standard {2} support:
When the header `<unistd.h>` is included, then the function prototype `gid_t getgid(void)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<unistd.h>` is included, then the function *getgid*() is declared with the result type *gid_t*. (See GA36 in 2.7.3.)

02(C) If *getgid*() is defined as a macro when the header `<unistd.h>` is included:
When the macro *getgid*() is invoked, then it expands to an expression with the result type *gid_t*. (See GA37 in 2.7.3.)

03(C) If *getgid*() is defined as a macro in the header `<unistd.h>`:
It protects its return value with extra parentheses when necessary. (See GA01 in 1.3.4.)

4.2.1.3.2 Description (4.2.1.2)

04(A) A call to *getgid*() returns the real group ID of the calling process.

4.2.1.3.3 Returns (4.2.1.3)

There are no assertions specific to this subclause.

4.2.1.3.4 Errors (4.2.1.4)

There are no assertions specific to this subclause.

4.2.1.4 *getegid*() (4.2.1)

4.2.1.4.1 Synopsis (4.2.1.1)

01(A) If the implementation provides C Standard {2} support:
When the header `<unistd.h>` is included, then the function prototype `gid_t getegid(void)` is declared. (see GA36 in 2.7.3.)

Otherwise:
When the header `<unistd.h>` is included, then the function *getegid*() is declared with the result type *gid_t*. (See GA36 in 2.7.3.)

02(C) If *getegid*() is defined as a macro when the header `<unistd.h>` is included:
When the macro *getegid*() is invoked, then it expands to an expression with the result type *gid_t*. (See GA37 in 2.7.3.)

03(C) If *getegid*() is defined as a macro in the header `<unistd.h>`:
It protects its return value with extra parentheses when necessary.(See GA01 in 1.3.4.)

4.2.1.4.2 Description (4.2.1.2)

04(A) A call to *getegid*() returns the effective group ID of the calling process.

4.2.1.4.3 Returns (4.2.1.3)

There are no assertions specific to this subclause.

4.2.1.4.4 Errors (4.2.1.4)

There are no assertions specific to this subclause.

4.2.2 Set User and Group IDs

4.2.2.1 *setuid*() (4.2.2)

4.2.2.1.1 Synopsis (4.2.2.1)

01(A) If the implementation provides C Standard {2} support:
When the header `<unistd.h>` is included, then the function prototype `int setuid(uid_t)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<unistd.h>` is included, then the function *setuid*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *setuid*() is defined as a macro when the header `<unistd.h>` is included:
When the macro *setuid*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *setuid*() is defined as a macro in the header `<unistd.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

4.2.2.1.2 Description (4.2.2.2)

04(C) If the behavior associated with {_POSIX_SAVED_IDS} is supported and the implementation provides the mechanism for creating processes with the appropriate privileges to change user IDs:

When the calling process has the required appropriate privileges to change user IDs, then a call to *setuid*(*uid*) sets the real user ID, effective user ID, and the saved set-user ID to *uid* and returns a value of zero.

05(C) If the behavior associated with {_POSIX_SAVED_IDS} is supported:

When the process does not have the required appropriate privileges to change user IDs, and argument uid is equal to the real user ID or the saved set-user ID, then a call to *setuid*(*uid*) sets the effective user ID to *uid* and returns a value of zero. The real user ID and saved set-user ID remain unchanged.

D01(C) If the implementation provides a method or methods for obtaining the appropriate privileges to change real, effective, and saved setuids:

The method(s) are documented in 2.2.2.4 or in 4.2.2.2 of the PCD.1.

06(C) If the behavior associated with {_POSIX_SAVED_IDS} is not supported and the implementation provides the mechanism for creating processes with the appropriate privileges to change user IDs:

When the calling process has the required appropriate privileges to change user IDs, then a call to *setuid*(*uid*) sets the real user ID and the effective user ID to *uid* and returns a value of zero.

07(C) If the behavior associated with {_POSIX_SAVED_IDS} is not supported:

When the process does not have the required appropriate privileges to change user IDs, and argument *uid* is equal to the real user ID, then a call to *setuid*(*uid*) sets the effective user ID to *uid* and returns a value of zero. The real user ID remains unchanged.

4.2.2.1.3 Returns (4.2.2.3)

R01 When a call to *setuid*() completes successfully, then a value of (*int*)0 is returned. (See Assertions 4–7 in 4.2.2.1.2.)

R02 When a call to *setuid*() completes unsuccessfully, then a value of (*int*)−1 is returned and sets *errno* to indicate the error. (See Assertions 8 and 9 in 4.2.2.1.4.)

4.2.2.1.4 Errors (4.2.2.4)

08(C) If the implementation supports an invalid value for uid:

When the value of the *uid* argument is invalid, then a call to *setuid*(*uid*) returns a value of (*int*)-1 and sets *errno* to [EINVAL]. Neither the real nor the effective user ID of the calling process is changed.

09(A) If the behavior associated with {_POSIX_SAVED_IDS} is supported:

When the process does not have the required appropriate privileges to change user IDs, and *uid* does not match either the real user ID or the saved set-user-ID, then a call to *setuid*(*uid*) returns a value of (*int*)-1 and sets *errno* to [EPERM]. Neither the real nor the effective user ID of the calling process is changed.

Otherwise:

When the process does not have the required appropriate privileges to change user IDs, and *uid* does not match the real user ID, then a call to *setuid*(*uid*) returns a value of (*int*)-1 and sets *errno* to [EPERM]. Neither the real nor the effective user ID of the calling process is changed.

4.2.2.2 *setgid*() (4.2.2)

4.2.2.2.1 Synopsis (4.2.2.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<unistd.h>` is included, then the function prototype `int setgid(gid_t)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<unistd.h>` is included, then the function *setgid()* is either declared with result type int or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *setgid*() is defined as a macro when the header `<unistd.h>` is included:

When the macro *setgid*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *setgid*() is defined as a macro in the header `<unistd.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

4.2.2.2.2 Description (4.2.2.2)

04(C) If the behavior associated with {_POSIX_SAVED_IDS} is supported and the implementation provides the mechanism for creating processes with the appropriate privileges to change group IDs:

When the calling process has the required appropriate privileges to change group IDs, then a call to *setgid*(*gid*) sets the real group ID, effective group ID, and the saved set-group ID to *gid* and returns a value of zero.

05(C) If the behavior associated with {_POSIX_SAVED_IDS} is supported:

When argument *gid* is equal to the real group ID or the saved set-group ID, then a call to *setgid*(*gid*) sets the effective group ID to *gid* and returns a value of zero. The real group ID, saved set-group ID, and any supplementary group IDs remain unchanged.

D01(C) If the implementation provides a method or methods for obtaining the appropriate privileges to change real and effective group IDs:

The method(s) are documented in 2.2.2.4 or in 4.2.2.2 of the PCD.1.

06(C) If the behavior associated with {_POSIX_SAVED_IDS} is not supported and the implementation provides the mechanism for creating processes with the appropriate privileges to change group IDs:

When a process has the required appropriate privileges to change group IDs then a call to *setgid*(*gid*) sets the real group ID and the effective group ID to *gid* and returns a value of zero.

07(C) If the behavior associated with (_POSIX_SAVED_IDS) is not supported:

When the process does not have the required appropriate privileges to change group IDs, and argument *gid* is equal to the real group ID, then a call to *setgid*(*gid*) sets the effective group ID to *gid* and returns a value of zero. The real group ID and any supplementary group IDs remain unchanged.

4.2.2.2.3 Returns (4.2.2.3)

R01 When a call to *setgid*() completes successfully, then a value of (*int*)0 is returned. (see Assertions 4–7 in 4.2.2.2.2.)

R02 When a call to *setgid*() completes unsuccessfully, then a value of (*int*)-1 is returned and sets *errno* to indicate the error. (see Assertions 8 and 9 in 4.2.2.2.4.)

4.2.2.2.4 Errors (4.2.2.4)

08(C) If the implementation supports an invalid value for *gid*:

When the value of gid argument is invalid, then a call to *setgid*(*gid*) returns a value of (*int*)-1 and sets *errno* to [EINVAL]. Neither the real nor the effective group ID of the calling process is changed.

09(A) If the behavior associated with {_POSIX_SAVED_IDS} is supported:

When the process does not have the required appropriate privileges to change group IDs, and *gid* does not match either the real group ID or the saved set-group-ID, then a call to *setgid*(*gid*) returns a value of (*int*)-1 and sets *errno* to [EPERM]. Neither the real nor the effective group ID of the calling process is changed.

Otherwise:

When the calling process does not have the required appropriate privileges to change group IDs, and *gid* does not match the real group ID, then a call to *setgid*(*gid*) returns a value of (*int*)-1 and sets *errno* to [EPERM]. Neither the real nor the effective group ID of the calling process is changed.

4.2.3 *getgroups*()

4.2.3.1 Synopsis

01(A) If the implementation provides C Standard {2} support:

When the header `<unistd.h>` is included, then the function prototype `int getgroups(int, gid_t *)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<unistd.h>` is included, then the function *getgroups*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *getgroups*() is defined as a macro when the header `<unistd.h>` is included:

When the macro *getgroups*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *getgroups*() is defined as a macro in the header `<unistd.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

4.2.3.2 Description

04(A) If {NGROUPS_MAX} > 0:

When the value of *gidsetsize* is greater than or equal to the number of supplementary group IDs of the process, then a call to *getgroups*(*gidsetsize*, *grouplist*) places the current list of supplementary group IDs (which may also contain the effective group ID of the calling process) into the array *grouplist* and returns the number of supplementary group IDs placed in *grouplist*.

Otherwise:

A call to *getgroups*(*gidsetsize*, *grouplist*) places no supplementary group IDs into the array *grouplist* and returns a value of zero.

D01(C) If the values of array entries with indices larger than or equal to the returned value from *getgroups*() is documented:

The details are contained in 4.2.3.2 of the PCD.1.

05(A) When *gidsetsize* is zero, then a call to *getgroups*(*gidsetsize*, *grouplist*) returns the number of supplementary group IDs (which may also contain the effective group ID of the calling process) associated with the calling process without modifying the array pointed to by the *grouplist* argument.

06(A) A call to *getgroups*(*gidsetsize*, *grouplist*) returns a value less than or equal to that returned by *sysconf*(*_SC_NGROUPS_MAX*).

4.2.3.3 Returns

R01 When a call to *getgroups*() completes successfully, then the number of supplementary group IDs is returned. (See Assertions 4 and 5 in 4.2.3.2.)

R02 When a call to *getgroups*() completes unsuccessfully, then a value of (*int*)−1 is returned and sets *errno* to indicate the error. (See Assertion 7 in 4.2.3.4.)

4.2.3.4 Errors

07(C) If a call to *sysconf*(*_SC_NGROUPS_MAX*) returns a value greater than zero:

When the parameter *gidsetsize* is not equal to zero and is less than the number of supplementary group IDs, then a call to *getgroups*(*gidsetsize*, *grouplist*) returns a value of (*int*)-1 and sets *errno* to [EINVAL].

4.2.4 *getlogin*()

4.2.4.1 Synopsis

01(A) If the implementation provides C Standard {2} support:

When the header `<unistd.h>` is included, then the function prototype `char * getlogin(void)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<unistd.h>` is included, then the function *getlogin*() is declared with the result type char *. (See GA36 in 2.7.3.)

02(C) If *getlogin*() is defined as a macro when the header `<unistd.h>` is included:

When the macro *getlogin*() is invoked, then it expands to an expression with the result type char *. (See GA37 in 2.7.3.)

03(C) If*getlogin*() is defined as a macro in the header `<unistd.h>`:

It protects its return value with extra parentheses when necessary. (See GA01 in 1.3.4.)

4.2.4.2 Description

04(A) A call to*getlogin*() returns a pointer to the login name of the user associated by the login activity with the controlling terminal.

Testing Requirements:

Test for the condition when the real user ID is equal to the effective user ID and when it is not equal to the effective user ID.

4.2.4.3 Returns

R01 When a call to *getlogin*() completes successfully, then a pointer to the login name of the user is returned. (See Assertion 4 in 4.2.4.2.)

05(B) When a call to *getlogin*() completes unsuccessfully because the login name of the user cannot be found, then a **NULL** pointer is returned.

See Reason 2 in Section 5. of POSIX.3 {4}.

D01(C) If the return value from *getlogin*() points to static data that may be overwritten by each *getlogin*() call, and this is documented:

The details are contained in 4.2.4.3 of the PCD.1

4.2.4.4 Errors

D02(C) If it is documented whether error conditions are detected for *getlogin*():
The details are contained in 4.2.4.4 of the PCD.1.

4.3 Process Groups

4.3.1 *getpgrp*()

4.3.1.1 Synopsis

01(A) If the implementation provides C Standard {2} support:
When the header `<unistd.h>` is included, then the function prototype `pid_t getpgrp(void)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<unistd.h>` is included, then the function *getpgrp*() is declared with the result type *pid_t*. (See GA36 in 2.7.3.)

02(C) If *getpgrp*() is defined as a macro when the header `<unistd.h>` is included:
When the macro *getpgrp*() is invoked, then it expands to an expression with the result type *pid_t*. (See GA37 in 2.7.3.)

03(C) If *getpgrp*() is defined as a macro in the header `<unistd.h>`:
It protects its return value with extra parentheses when necessary. (See GA01 in 1.3.4.)

4.3.1.2 Description

04(A) A call to *getpgrp*() returns the process group ID of the calling process.

4.3.1.3 Returns

There are no assertions specific to this subclause.

4.3.1.4 Errors

There are no assertions specific to this subclause.

4.3.2 *setsid*()

4.3.2.1 Synopsis

01(A) If the implementation provides C Standard {2} support:
When the header `<unistd.h>` is included, then the function prototype `pid_t setsid(void)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<unistd.h>` is included, then the function *setsid*() is declared with the result type *pid_t*. (See GA36 in 2.7.3.)

02(C) If *setsid*() is defined as a macro when the header `<unistd.h>` is included:
When the macro *setsid*() is invoked, then it expands to an expression with the result type *pid_t*. (See GA37 in 2.7.3.)

03(C) If *setsid*() is defined as a macro in the header `<unistd.h>`:
It protects its return value with extra parentheses when necessary. (See GA01 in 1.3.4.)

4.3.2.2 Description

04(A) When the calling process is not a process group leader, then a call to *setsid*() creates a new session, and the value of the process group ID of the calling process is returned.

05(A) The new session created with a call to *setsid*() has the calling process as the session leader. On return from a successful call to *setsid*(), the calling process is the process group leader of a new process group.

06(PCTS_GTI_DEVICE?A:UNTESTED)

When a call to *setsid*() completes successfully, then on return the process has no controlling terminal.

07(A) A call to *setsid*() sets the process group ID of the calling process to the process ID of the calling process.

R01 When a call to *setsid*() completes successfully, then on return the calling process is the only process in the new process group and this calling process is the only process in the new session. (See Assertions 4 and 5 in 4.3.2.2.)

4.3.2.3 Returns

R02 When a call to *setsid*() completes successfully, then the value of the process group ID of the calling process is returned. (See Assertion 4 in 4.3.2.2.)

R03 When a call to *setsid*() completes unsuccessfully, then a value of (*pid_t*)–1 is returned and sets *errno* to indicate the error. (See Assertions 8 and 9 in 4.3.2.4.)

4.3.2.4 Errors

08(PCTS_GTI_DEVICE?A:UNTESTED)

When the calling process is already a process group leader, then a call to *setsid*() returns a value of (*pid_t*)-1 and sets *errno* to [EPERM]. The calling process does not relinquish its controlling terminal.

09(PCTS_GTI_DEVICE?B:UNTESTED)

When the process group ID of a process other than the calling process matches the process ID of the calling process, then a call to *setsid*() returns a value of (*pid_t*)-1 and sets *errno* to [EPERM]. The calling process does not relinquish its controlling terminal.

See Reason 3 in Section 5. of POSIX.3 {4].

4.3.3 *setpgid*()

4.3.3.1 Synopsis

01(A) If the implementation provides C Standard {2} support:
When the header `<unistd.h>` is included, then the function prototype
`int setpgid(pid_t, pid_t)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<unistd.h>` is included, then the function *setpgid*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *setpgid*() is defined as a macro when the header `<unistd.h>` is included:
When the macro *setpgid*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *setpgid*() is defined as a macro in the header `<unistd.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

4.3.3.2 Description

D01(C) If {_POSIX_JOB_CONTROL} is not defined and it is documented whether *setpgid*() is supported:
The documentation is in 4.3.3.2 of the PCD.1.

04(C) If either the behavior associated with (_POSIX_JOB_CONTROL} or *setpgid*() is supported:
When *pid* refers to a process that is not a session leader, *pgid* and *pid* are not equal, *pgid* matches the process group ID of a process in the same session as the calling process, *pid* matches the process ID of a process in the same session as the calling process, and *pid* matches the process ID of a child process of the calling process that has not executed an `exec` function, then a call to *setpgid*() sets the process group ID of the indicated process to *pgid* and returns a value of zero.

05(C) If either the behavior associated with {_POSIX_JOB_CONTROL} or *setpgid*() is supported:
When *pid* refers to a process that is not a session leader, *pgid* and *pid* are equal, *pid* matches the process ID of a process in the same session as the calling process, and *pid* matches the process ID of a child process of the calling process that has not executed an `exec` function, then a call to *setpgid*() sets the process group ID of the indicated process to *pgid* and returns a value of zero.

06(C) If either the behavior associated with {_POSIX_JOB_CONTROL} or *setpgid*() is supported:
When *pid* refers to a process that is not a session leader, *pgid* matches the process group ID of a process in the same session as the calling process, and *pid* is equal to zero, then a call to *setpgid*() sets the process group ID of the calling process to *pgid* and returns a value of zero.

07(C) If either the behavior associated with {_POSIX_JOB_CONTROL} or *setpgid*() is supported:
When *pid* refers to a process that is not a session leader, *pid* matches the process ID of a process in the same session as the calling process, *pgid* is equal to zero, and *pid* matches the process ID of a child process of the calling process that has not executed an `exec` function, then a call to *setpgid*() sets the process group ID of the target process to *pid* and returns a value of zero.

08(C) If either the behavior associated with {_POSIX_JOB_CONTROL} or *setpgid*() is supported:
When the *pid* of the calling process is not a session leader, and when both the *pid* and *pgid* are set to zero, then a call to *setpgid*() sets the group ID of the calling process equal to its process ID and returns a value of zero.

R01 A call to *setpgid*() returns a value of (*int*)–1 and sets *errno* to [ENOSYS]. (See Assertion 12 in 4.3.3.4.)

4.3.3.3 Returns

R02 When a call to *setpgid*() completes successfully, then a value of (*int*)0 is returned. (See Assertions 4–8 in 4.3.3.2.)

R03 When a call to *setpgid*() completes unsuccessfully, then a value of (*int*)–1 is returned and sets *errno* to indicate the error. (See Assertions 9–16 in 4.3.3.4.)

4.3.3.4 Errors

09(C) If either the behavior associated with {_POSIX_JOB_CONTROL} or *setpgid*() is supported:
When the value of the *pid* argument matches the process ID of a child process of the calling process, and when the child process has successfully executed an `exec`() function, then a call to set *pgid*(*pid*, *pgid*) returns a value of (*int*)-1 and sets *errno* to [EACCES]. The process group ID of the calling process is not changed.

10(C) If either the behavior associated with {_POSIX_JOB_CONTROL} or *setpgid*() is supported:

When the value of the *pgid* argument is less than zero, then a call to set *pgid*(*pid*, *pgid*) returns a value of (*int*)-1 and sets *errno* to [EINVAL]. The process group ID of the calling process is not changed.

11(C) If either the behavior associated with {_POSIX_JOB_CONTROL} or *setpgid*() is supported and the implementation supports an invalid value when *pgid* is greater than 0:
When the value of the *pgid* argument is invalid, then a call to set *pgid*(*pid*, *pgid*) returns a value of (*int*)-1 and sets *errno* to [EINVAL]. The process group ID of the calling process is not changed.

12(C) If the behavior associated with {_POSIX_JOB_CONTROL} is not supported and *setpgid*() is not supported:
A call to *setpgid*() returns a value of (*int*)-1 and sets *errno* to [ENOSYS]. The process group ID of the calling process is not changed.

13(C) If either the behavior associated with {_POSIX_JOB_CONTROL} or *setpgid*() is supported:
When the value of the *pid* argument matches the process ID of a session leader, then a call to set*pgid*(*pid*, *pgid*) returns a value of (*int*)-1 and sets *errno* to [EPERM]. The process group ID of the calling process is not changed.

14(C) If either the behavior associated with {_POSIX_JOB_CONTROL} or *setpgid*() is supported:
When the value of the *pid* argument matches the process ID of a child process of the calling process, and when the child process is not in the same session as the calling process, then a call to *setpgid*(*pid*, *pgid*) returns a value of (*int*)-1 and sets *errno* to [EPERM]. The process group ID of the calling process is not changed.

15(C) If either the behavior associated with {_POSIX_JOB_CONTROL} or *setpgid*() is supported:
When the value of the *pgid* argument does not match the process ID of the process indicated by the pid argument, and when there is no process with a process group ID that matches the value of pgid argument in the same session as the calling process, then a call to *setpgid*(*pid*, *pgid*) returns a value of (*int*)-1 and sets *errno* to [EPERM]. The process group ID of the calling process is not changed.

16(C) If either the behavior associated with {_POSIX_JOB_CONTROL} or *setpgid*() is supported:
When the value of the pid argument does not match the process ID of the calling process or a child process of the calling process, then a call to *setpgid*(*pid*, *pgid*) returns a value of (*int*)-1 and sets *errno* to [ESRCH]. The process group ID of the calling process is not changed.

4.4 System Identification

4.4.1 *uname*()

4.4.1.1 Synopsis

01(A) If the implementation provides C Standard {2} support:
When the header `<sys/utsname.h>` is included, then the function prototype `int uname(struct utsname *)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<sys/utsname.h>` is included, then the function *uname*() is either declared with result type int or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *uname*() is defined as a macro when the header `<sys/utsname.h>` is included:
When the macro *uname*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *uname*() is defined as a macro in the header `<sys/utshame.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

4.4.1.2 Description

04(A) A call to *uname*(name) stores information identifying the current operating system in a structure pointed to by name and returns a nonnegative value.

05(A) When the header `<sys/utsname.h>` is included, then the data items *sysname*, *nodename*, *release*, *version*, and *machine* are members of *struct utsname*.

06(A) When a call to *uname*() is successful, then the *sysname*, *nodename*, *release*, *version*, and *machine* elements of the *utsname* structure referenced by name are each null-terminated character arrays.

D01(A) The format of each member of *struct utsname* defined in Table 4-1 of POSIX.1 {3} is described in 4.4.1.2 of the PCD.1.

4.4.1.3 Returns

R01 When a call to *uname*() completes successfully, then a nonnegative value is returned. (See Assertion 4 in 4.4.1.2.)

07(B) When a call to *uname*() completes unsuccessfully, then a value of (*int*)–1 is returned and sets *errno* to indicate the error.

See Reason 2 in Section 5. of POSIX.3 {4}.

4.4.1.4 Errors

D02(C) If error conditions are detected for *uname*(), and this is documented:
The details are contained in 4.4.1.4 of the PCD.1.

4.5 Time

4.5.1 *time*()

4.5.1.1 Synopsis

01(A) If the implementation provides C Standard {2} support:
When the header `<time.h>` is included, then the function prototype `time_time(time_t *)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<time.h>` is included, then the function *time*() is declared with the result type *time_t*. (See GA36 in 2.7.3.)

02(C) If *time*() is defined as a macro when the header `<time.h>` is included:
When the macro *time*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *time_t*. (See GA37 in 2.7.3.)

03(C) If *time*() is defined as a macro in the header `<time.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

4.5.1.2 Description

04(A) When *tloc* is NULL or when *tloc* is a valid address, then a call to *time*(*tloc*) returns the value of time in seconds since the Epoch.

05(A) When *tloc* is other than a **NULL** pointer, then a call to *time*(*tloc*) causes the return value to be stored in the location pointed to by *tloc*.

4.5.1.3 Returns

R01 When a call to *time*() completes successfully, then the value of time is returned. (See Assertions 4 and 5 in 4.5.1.2.)

06(B) When a call to *time*() completes unsuccessfully, then a value of (*time_t*)-1 is returned and sets *errno* to indicate the error.

See Reason 2 in Section 5. of POSIX.3 {4}.

4.5.1.4 Errors

D01(c) If the implementation documents whether and under what conditions errors are detected for *time*():
The details are contained in 4.5.1.4 of the PCD.1.

4.5.2 *times*()

4.5.2.1 Synopsis

01(A) If the implementation provides C Standard {2} support:
When the header `<sys/times.h>` is included, then the function prototype `clock_t times(struct tms *)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<sys/times.h>` is included, then the function *times*() is declared with the result type *clock_t*. (See GA36 in 2.7.3.)

02(C) If *times*() is defined as a macro when the header `<sys/times.h>` is included:
When the macro *times*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *clock_t*. (See GA37 in 2.7.3.)

03(C) If *times*() is defined as a macro in the header `<sys/times.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

4.5.2.2 Description

04(A) When the header `<sys/times.h>` is included, then the *structure tms* is defined and contains at least the members *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* of type *clock_t*.

R01 All members of the structure pointed to by buffer, after a call to *times*(*buffer*), contains information reported in units of clock ticks. (See Assertions 6–9 in 4.5.2.2.)

05(A) When a call to *times*() completes successfully, then the elapsed real time in clock ticks from an arbitrary point in the past is returned.

R02 A call to *times*(*buffer*) from a parent process fills the structure pointed to by buffer with accounting information that contains the *tms_cutime* and *tms_cstime* times for processes for which *wait*() or *waitpid*() have been called, and that does not contain *tms_cutime* and *tms_cstime* times for those processes for which the call to *wait*() or *waitpid*() has not waited. (See Assertions 8 and 9 in 4.5.2.2.)

06(A) A call to *times*(*buffer*) fills the structure pointed to by buffer with accounting information, and the value *tms_utime* in clock ticks is the CPU time charged for the execution of user instructions while executing instructions of the calling process.

07(A) A call to *times*(*buffer*) fills the structure pointed to by buffer with accounting information, and the value *tms_stime* in clock ticks is the CPU time charged for the execution by the system on behalf of the calling process.

08(A) A call to *times*(*buffer*) fills the structure pointed to by buffer with accounting information that contains the *tms_cutime*, in clock ticks, which is the sum of the *tms_utime* and *tms_cutime* of all terminated child processes for which *wait*() or *waitpid*() has been called, and that does not contain the *tms_utime* and *tms_cutime* times for those processes for which the call to *wait*() or *waitpid*() has not waited.

09(A) A call to *times*(*buffer*) fills the structure pointed to by buffer with accounting information that contains the *tms_cstime* in clock ticks, which is the sum of the *tms_stime* and *tms_cstime* of all terminated child processes for which *wait*() or *waitpid*() has been called, and that does not contain the *tms_stime* and *tms_cstime* times for those processes for which the call to *wait*() or *waitpid*() has not waited.

4.5.2.3 Returns

R03 When a call to *times*() completes successfully, then the elapsed real time in clock ticks from an arbitrary point in the past is returned. (See Assertion 5 in 4.5.2.2.)

10(A) A call to *times*() keeps the arbitrary point in the past as a constant between invocations of *times*() within the same process.

11(B) When a call to *times*() completes unsuccessfully, then a value of (*clock_t*)-1 is returned and sets *errno* to indicate the error.

See Reason 2 in Section 5. of POSIX.3 {4}.

D01(C) If it is documented whether the return value from *times*() may overflow the range of type *clock_t*:
The details are contained in 4.5.2.3 of the PCD.1.

4.5.2.4 Errors

D02(C) If the implementation documents whether and under what conditions errors are detected for *times*():
The details are contained in 4.5.2.4 of the PCD.1.

4.6 Environment Variables

4.6.1 *getenv*()

4.6.1.1 Synopsis

01(A) If the implementation provides C Standard {2} support:
When the header `<stdlib.h>` is included, then the function prototype
`char * qetenv(const char *)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<stdlib.h>` is included, then the function *getenv*() is declared with the result type *char* *. (See GA36 in 2.7.3.)

02(C) If *getenv*() is defined as a macro when the header `<stdlib.h>` is included:
When the macro *getenv*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *char* *. (See GA37 in 2.7.3.)

03(C) If *getenv*() is defined as a macro in the header `<stdlib.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

4.6.1.2 Description

04(A) A call to *getenv*(*name*) returns a pointer to the value associated with the environment variable name.

05(A) A call to *getenv*(*name*) returns a **NULL** pointer for names that do not exist in the environment.

06(A) Upper- and lowercase letters in the environment retain their unique identities.

07(A) Environment variable names support the portable filename character set.

4.6.1.3 Returns

R01 When a call to *getenv*() completes successfully, then either a pointer to a string containing the environment variable is returned or, if the environment variable requested cannot be found, a **NULL** pointer is returned. (See Assertions 4 and 5 in 4.6.1.2.)

08(D) If the implementation supports error conditions for *getenv*():
When a call to *getenv*() completes unsuccessfully, then a **NULL** pointer is returned to indicate the error.

See Reason 2 in Section 5. of POSIX.3 {4}.

D01(C) If it is documented whether the return value from *getenv*() points to static data that may be overwritten by subsequent calls to *getenv*():
The details are contained in 4.6.1.3 of the PCD.1.

4.6.1.4 Errors

D02(C) If the implementation documents whether and under what conditions errors are detected for *getenv*():
The details are contained in 4.6.1.4 of the PCD.1.

4.7 Terminal Identification

4.7.1 *ctermid*()

4.7.1.1 Synopsis

01(A) If the implementation provides C Standard {2} support:
When the header `<stdio.h>` is included, then the function prototype `char * ctermid(char *)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<stdio.h>` is included, then the function *ctermid*() is declared with the result type *char* *. (See GA36 in 2.7.3.)

02(C) If *ctermid*() is defined as a macro when the header `<stdio.h>` is included:
When the macro *ctermid*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *char* *. (See GA37 in 2.7.3.)

03(C) If *ctermid*() is defined as a macro in the header `<stdio.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

4.7.1.2 Description

04(PCTS_GTI_DEVICE?A:UNTESTED)

A call to *ctermid*() generates a string that, when used. as a pathname, refers to the current controlling terminal for the current process.

05(PCTS_GTI_DEVICE?A:UNTESTED)

When a call to *ctermid*(*(char *) 0*) completes successfully, then a pointer to a string is returned that, when used as a pathname, refers to the current controlling terminal of the process.

4.7.1.3 Returns

06(A) When the header `<stdio.h>` is included, then the symbolic constant L_ctermid is defined and has a value greater than zero.

07(PCTS_GTI_DEVICE?A:UNTESTED)

When a call to *ctermid*(s) completes successfully and s is not a **NULL** pointer, then a string that, when used as a pathname, refers to the current controlling terminal is placed in the character array to which s points, the length of the string pointed to by s is less than L_ctermid bytes, and the value of s is returned.

08(B) When a call to *ctermid*() completes unsuccessfully, then an empty string is returned.

See Reason 2 in Section 5. of POSIX.3 {4}.

D01(C) If it is documented whether the argument to *ctermid*() is a **NULL** pointer when the return value from *ctermid*() points to static data that may be overwritten by subsequent calls to *ctermid()*:
The details are contained in 4.7.1.3 of the PCD.1.

4.7.1.4 Errors

D02(C) If the implementation documents whether and under what conditions errors are detected for *ctermid*():
The details are contained in 4.7.1.4 of the PCD.1.

4.7.2 Determinable Terminal Device Name

4.7.2.1 *ttyname*() (4.7.2)

4.7.2.1.1 Synopsis (4.7.2.1)

01(A) If the implementation provides C Standard {2} support:
When the header `<unistd.h>` is included, then the function prototype `char * ttyname(int)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<unistd.h>` is included, then the function *ttyname*() is declared with the result type *char* *. (See GA36 in 2.7.3.)

02(C) If *ttyname*() is defined as a macro when the header `<unistd.h>` is included:
When the macro *ttyname*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *char* *. (See GA37 in 2.7.3.)

03(C) If *ttyname*() is defined as a macro in the header `<unistd.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

4.7.2.1.2 Description (4.7.2.2)

04(PCTS_GTI_DEVICE?A:UNTESTED)

A call to *ttyname*(*fildes*) returns a pointer to a string containing a null-terminated pathname for the terminal associated with file descriptor *fildes*.

D01(C) If it is documented whether the return value from *ttyname*() points to static data that may be overwritten by subsequent calls to *ttyname*():

The details are contained in 4.7.2.2 of the PCD.1.

4.7.2.1.3 Returns (4.7.2.3)

05(A) When the file descriptor argument is not a valid file descriptor associated with a terminal device, then a call to *ttyname*(*fildes*) completes unsuccessfully, and a **NULL** pointer is returned.

06(B) When the pathname cannot be determined, then a call to *ttyname*(*fildes*) completes unsuccessfully, and a **NULL** pointer is returned.

See Reason 3 in Section 5. of POSIX.3 [4].

4.7.2.1.4 Errors (4.7.2.4)

D02(C) If the implementation documents whether and under what conditions errors are detected for *ttyname*():

The details are contained in 4.7.2.4 of the PCD.1.

4.7.2.2 *isatty*() (4.7.2)

4.7.2.2.1 Synopsis (4.7.2.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<unistd.h>` is included, then the function prototype `int isatty(int)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<unistd.h>` is included, then the function *isatty*() is either declared with result type int or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *isatty*() is defined as a macro when the header `<unistd.h>` is included:

When the macro *isatty*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *isatty*() is defined as a macro in the header `<unistd.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

4.7.2.2.2 Description (4.7.2.2)

04(PCTS_GTI_DEVICE?A:UNTESTED)

When the argument *fildes* is a valid file descriptor associated with a terminal, then a call to *isatty*(*fildes*) returns a vale of 1.

05(A) When the argument *fildes* is not a file descriptor associated with a terminal, then a call to *isatty*(*fildes*) returns a value of 0.

4.7.2.2.3 Returns (4.7.2.3)

There are no assertions specific to this subclause.

4.7.2.2.4 Errors (4.7.2.4)

D01(C) If the implementation documents whether and under what conditions errors are detected for *isatty*():
The details are contained in 4.7.2.4 of the PCD.1.

4.8 Configurable System Variables

4.8.1 *sysconf*() (4.8.1)

4.8.1.1 Synopsis

01(A) If the implementation provides C Standard {2} support:
When the header `<unistd.h>` is included, then the function prototype `long sysconf(int)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<unistd.h>` is included, then the function *sysconf*() is declared with the result type *long*. (See GA36 in 2.7.3.)

02(C) If *sysconf*() is defined as a macro when the header `<unistd.h>` is included:
When the macro *sysconf*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *long*. (See GA37 in 2.7.3.)

03(C) If *sysconf*() is defined as a macro in the header `<unistd.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

4.8.1.2 Description

04(A) When the header `<unistd.h>` is included, then the symbolic constants _SC_ARG_MAX, _SC_CHILD_MAX, _SC_CLK_TCK, _SC_NGROUPS_MAX, _SC_OPEN_MAX, _SC_JOB_CONTROL, _SC_SAVED_IDS, _SC_VERSION, _SC_STREAM_MAX, and _SC_TZNAME_MAX are defined and are distinct.

05(A) If ARG_MAX is defined when the header `<limits.h>` is included:
A call to *sysconf*(*_SC_ARG_MAX*) either returns -1 without changing *errno* or returns a value greater than or equal to {ARG_MAX}.

Otherwise:
A call to *sysconf*(*_SC_ARG_MAX*) either returns -1 without changing *errno* or returns a value greater than or equal to {_POSIX_ARG_MAX}.

06(A) If CHILD_MAX is defined when the header `<limits.h>` is included:
A call to *sysconf*(*_SC_CHILD_MAX*) either returns -1 without changing *errno* or returns a value greater than or equal to {CHILD_MAX}.

Otherwise:
A call to *sysconf*(*_SC_CHILD_MAX*) either returns -1 without changing *errno* or returns a value greater than or equal to {_POSIX_CHILD_MAX}.

07(A) A call to *sysconf*(*_SC_CLK_TCK*) returns a value equal to the value returned by the macro CLK_TCK, defined in `<time.h>`.

08(A) A call to *sysconf*(*_SC_NGROUPS_MAX*) returns a value greater than or equal to the value of NGROUPS_MAX in `<limits.h>`.

09(A) If OPEN_MAX is defined when the header `<limits.h>` is included:
A call to *sysconf*(*_SC_OPEN_MAX*) either returns -1 without changing *errno* or returns a value greater than or equal to {OPEN_MAX}.

Otherwise:
A call to *sysconf*(*_SC_OPEN_MAX*) either returns -1 without changing *errno* or returns a value greater than or equal to {_POSIX_OPEN_MAX}.

10(A) If STREAM_MAX is defined when the header `<limits.h>` is included:
A call to *sysconf*(*_SC_STREAM_MAX*) either returns -1 without changing *errno* or returns a value greater than or equal to {STREAM_MAX}.

Otherwise:
A call to *sysconf*(*_SC_STREAM_MAX*) either returns -1 without changing *errno* or returns a value greater than or equal to {_POSIX_STREAM_MAX}.

11(A) If TZNAME_MAX is defined when the header `<limits.h>` is included:
A call to *sysconf*(*_SC_TZNAME_MAX*) either returns -1 without changing *errno* or returns a value greater than or equal to {TZNAME_MAX}.

Otherwise:
A call to *sysconf*(*_SC_TZNAME_MAX*) either returns -1 without changing *errno* or returns a value greater than or equal to {_POSIX_TZNAME_MAX}.

12(A) If {_POSIX_JOB_CONTROL} is defined when the header `<unistd.h>` is included:
A call to *sysconf*(*_SC_JOB_CONTROL*) returns a value other than -1.

Otherwise:
A call to *sysconf*(*_SC_JOB_CONTROL*) does not change the value of *errno*.

13(A) If {_POSIX_SAVED_IDS} is defined when the header `<unistd.h>` is included:
A call to *sysconf*(*_SC_SAVED_IDS*) returns a value other than -1.

Otherwise:
A call to *sysconf*(*_SC_SAVED_IDS*) does not change the value of *errno*.

14(A) A call to *sysconf*(*_SC_VERSION*) returns a value equal to {_POSIX_VERSION}.

D01(C) If the implementation supports variables beyond those listed in Table 4-2 of POSIX.1 {3}, and this is documented:
The details are contained in 4.8.1.2 of the PCD.1.

4.8.1.3 Returns

R01 When a call to *sysconf*(*name*) completes unsuccessfully because argument name is an invalid value, then a value of (*long*)-1 is returned. (See Assertion 18 in 4.8.1.4.)

15(C) If the behavior associated with {_POSIX_JOB_CONTROL} is not supported:
A call to *sysconf*(*_SC_JOB_CONTROL*) returns a value of (*long*)-1, and the value of *errno* is not changed.

16(C) If the behavior associated with {_POSIX_SAVED_IDS} is not supported:
A call to *sysconf*(*_SC_SAVED_IDS*) returns a value of (*long*)-1, and the value of *errno* is not changed.

R02 When a call to *sysconf*(*name*) completes successfully and the variable corresponding to the argument *name* is defined on the system, then the current variable value on the system is returned. (See Assertions 5–14 in 4.8.1.2.)

17(B) When a call to *sysconf*(*name*) completes successfully, then the value returned does not change during the lifetime of the calling process.

See Reason 3 in Section 5. of POSIX.3 {4}.

4.8.1.4 Errors

18(A) When the value of the argument *name* is invalid, then a call to *sysconf*(*name*) returns a value of(*long*)-1 and sets *errno* to [EINVAL].

4.8.2 Special Symbol {CLK_TCK} (4.8.1.5)

R03 The special symbol CLK_TCK yields the same result as a call to *sysconf*(*_SC_CLK_TCK*). (See Assertion 7 in 4.8.1.2.)

19(A) When the header `<time.h>` is included, then the special symbol CLK_TCK is defined.

5. Files and Directories

5.1 Directories

5.1.1 Format of Directory Entries

01(A) When the header `<dirent.h>` is included, then the structure *dirent* is defined.

D01(C) If the internal format of directories is documented:
The details are contained in 5.1.1 of the PCD.1.

02(A) The *struct dirent* includes the member *d_name* whose type is *char*[].

D02(C) If the size of the array *d_name* is documented:
The details are contained in 5.1.1 of the PCD.1.

03(B) When a call to *readdir*() completes successfully, then the number of bytes preceding the null character in the array *d_name* does not exceed {NAME_MAX} bytes.

See Reason 1 in Section 5. of POSIX.3 {4}.

5.1.2 Directory Operations

5.1.2.1 *opendir*() (5.1.2)

5.1.2.1.1 Synopsis (5.1.2.1)

01(A) If the implementation provides C Standard {2} support:
When the header `<dirent.h>` is included, then the function prototype
`DIR * opendir(const char *)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<dirent.h>` is included, then the function *opendir*() is declared with the result type *DIR* *. (See GA36 in 2.7.3.)

02(C) If *opendir*() is defined as a macro when the header `<dirent.h>` is included:
When the macro *opendir*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *DIR* *. (See GA37 in 2.7.3.)

03(C) If *opendir*() is defined as a macro in the header `<dirent.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

5.1.2.1.2 Description (5.1.2.2)

04(A) When the header `<dirent.h>` is included, then the type *DIR* is defined.

R01 When an *exec* function is called, then the directory stream is closed in the new process image. (See Assertion 38 in 3.1.2.2.)

05(A) When the first filename component of the *dirname* argument is ".", and the pathname does not begin with a slash, then *opendir*() resolves the pathname by locating the second filename component (when specified) in the current working directory. (See GA14 in 2.3.6.)

06(A) When the *dirname* argument points to the string "`/`", then *opendir*() resolves the pathname to the root directory of the process. (See GA15 in 2.3.6.)

07(A) When the *dirname* argument points to the string "`///`", then *opendir*() resolves the pathname to the root directory of the process. (See GA16 in 2.3.6.)

08(A) When the *dirname* argument points to a string beginning with a single slash or beginning with three or more slashes, then *opendir*() resolves the pathname by locating the first filename component of the pathname in the root directory of the process. (see GA17 in 2.3.6.)

09(A) When the first filename component of the *dirname* argument is "..", the pathname does not begin with a slash, and the current working directory is not the root directory of the process, then *opendir*() resolves the pathname by locating the second filename component (when specified) in the parent directory of the current working directory. (See GA18 in 2.3.6.)

10(A) When the *dirname* argument points to the string "F1/" and `F1` is a directory, then *opendir*() resolves the pathname to `F1`, which is in the current working directory. (See GA19 in 2.3.6.)

11(A) When the argument *dirname* points to the string "F1//" and `F1` is a directory, then *opendir*() resolves the pathname to `F1`, which is in the current working directory. (See GA20 in 2.3.6.)

12(A) When the *dirname* argument points to the string "F1/F2", then *opendir*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA21 in 2.3.6.)

13(A) When the pathname argument points to the string " F1/./F2", then *opendir*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA22 in 2.3.6.)

14(A) When the *dirname* argument points to the string "F1/../F1/F2", then *opendir*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (see GA23 in 2.3.6.)

15(A) When the *dirname* argument points to the string "F1//F2", then *opendir*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA24 in 2.3.6.)

16(C) If {_POSIX_NO_TRUNC} is not supported in the corresponding directory:
When the *dirname* component is a string of more than {NAME_MAX} bytes in a directory for which {_POSIX_NO_TRUNC} is not supported, then *opendir*() resolves the pathname component by truncating it to {NAME_MAX} bytes. (See GA25 in 2.3.6.)

17(A) On a call to *opendir*(*dirname*), the pathname *dirname* supports filenames containing any of the following characters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9. _ -

(See GA02 in 2.2.2.32.)

18(A) On a call to *opendir*(*dirname*), the pathname *dirname* retains the unique identity of its upper- and lowercase letters. (See GA03 in 2.2.2.32.)

19(A) When *opendir*(*dirname*) is granted read access to *dirname* and search access to each component of the path prefix of *dirname*, then the standard file access control mechanism does not cause *opendir*() to indicate that file access is denied.

20(A) A call to *opendir*(*dirname*) opens a directory stream corresponding to the directory named by the *dirname* argument.

21(A) A call to *opendir*() positions the directory stream at the first directory entry.

D01(C) If the action taken when a file is removed from or added to the directory after the most recent call to *opendir*() is documented:

The details on whether a subsequent call to *readdir*() returns an entry for that file is contained in 5.1.2.2 of the PCD.1.

5.1.2.1.3 Returns (5.1.2.3)

R02 When a call to *opendir*() completes successfully, then an object of type *DIR* * is returned. (See Assertions 5–19 in 5.1.2.1.2.)

R03 When a call to *opendir*() completes unsuccessfully, then a value of NULL is returned and sets *errno* to indicate the error. (See Assertions 22–24, 26–29, 31, and 33 in 5.1.2.1.4.)

5.1.2.1.4 Errors (5.1.2.4)

22(A) When search permission is denied for any component of the path prefix of *dirname*, then a call to *opendir*(*dirname*) returns a value of NULL and sets *errno* to [EACCES].

23(A) When read permission is denied for *dirname*, then a call to *opendir*(*dirname*) returns a value of NULL and sets *errno* to [EACCES].

24({NAME_MAX}≤{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of *dirname* is longer than the maximum number of bytes in a filename {NAME_MAX}, then a call to *opendir*(*dirname*) returns a value of NULL and sets *errno* to [ENAMETOOLONG].

25({NAME_MAX}>{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of *dirname* equals {PCTS_NAME_MAX}, then a call to *opendir*(*dirname*) succeeds.

26(A) If {PATH_MAX}≤{PCTS_PATH_MAX}:

When the length of the *dirname* argument exceeds the maximum number of bytes in a pathname {PATH_MAX}, then a call to *opendir*(*dirname*) returns a value of NULL and sets *errno* to [ENAMETOOLONG].

Otherwise:

When the length of the path argument is {PCTS_PATH_MAX}, then a call to *opendir*() is successful.

27(A) When the argument *dirname* is not an existing directory, then a call to *opendir*(*dirname*) returns a value of NULL and sets *errno* to [ENOENT].

28(A) When the argument *dirname* points to an empty string, then a call to *opendir*(*dirname*) returns a value of NULL and sets *errno* to [ENOENT].

29(A) When a component of the directory argument *dirname* is not a directory, then a call to *opendir*(*dirname*) returns a value of NULL and sets *errno* to [ENOTDIR].

D02(C) If the implementation supports the detection of [EMFILE] for *opendir*():

The details under which [EMFILE] occurs for *opendir*() are contained in 5.1.2.4 of the PCD.1. (See DGA02 in 2.4.)

30({OPEN_MAX}>{PCTS_OPEN_MAX}?A:UNTESTED)

{PCTS_OPEN_MAX} files or directories can be opened.

31({OPEN_MAX}≤{PCTS_OPEN_MAX}?C:UNTESTED)

If {PCD_TYPE_DIR} is **TRUE** and the implementation supports the detection of [EMFILE] for *opendir*():

When {OPEN_MAX} file descriptors have been opened, then a subsequent call to *opendir* (*dirname*) returns a value of NULL and sets *errno* to [EMFILE].

32({OPEN_MAX}≤{PCTS_OPEN_MAX}?D:UNTESTED)

If {PCD_TYPE_DIR} is not documented and the implementation supports the detection of [EMFILE] for *opendir*():

When {OPEN_MAX} file descriptors have been opened, then a subsequent call to *opendir* (*dirname*) will either succeed, returning a value of non-NULL, or will fail, returning a value of NULL and setting *errno* to [EMFILE].

See Reason 2 in Section 5. of POSIX.3 {4}.

33({OPEN_MAX}≤{PCTS_OPEN_MAX}?D:UNTESTED)

If {PCD_TYPE_DIR} is **TRUE** and the implementation does not support the detection of [EMFILE] for *opendir*():

When [OPEN_MAX] files have been opened, then a subsequent call to *opendir*(*dirname*) is successful (unless a different error condition is detected). (See GA26 in 2.4.)

See Reason 2 in Section 5. of POSIX.3 {4}.

D03(C) If the implementation supports the detection of [ENFILE] for *opendir*():

The details under which [ENFILE] occurs for *opendir*() are contained in 5.1.2.4 of the PCD.1. (See DGA02 in 2.4.)

34(system limit on open files is ≤ {PCTS_OPEN_MAX}?D:UNTESTED)

If {PCD_TYPE_DIR} is **TRUE** and the implementation supports the detection of [ENFILE] for *opendir*():

When the limit on the number of open files on the system has been opened, then a subsequent call to *opendir* (*dirname*) returns a value of NULL and sets *errno* to [ENFILE].

See Reason 4 in Section 5. of POSIX.3 {4}.

35(system limit on open files is ≤ {PCTS_OPEN_MAX}? D:UNTESTED)

If {PCD_TYPE_DIR} is **TRUE** and the implementation does not support the detection of [ENFILE] for *opendir*():

When the limit on the number of open files on the system has been opened, then a subsequent call to *opendir* (*dirname*) is successful (unless a different error condition is detected). (See GA26 in 2.4.)

See Reason 2 in Section 5. of POSIX.3 {4}.

5.1.2.2 *readdir*() (5.1.2)

5.1.2.2.1 Synopsis (5.1.2.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<dirent.h>` is included, then the function prototype `struct dirent * readdir(DIR *)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<dirent.h>` is included, then the function *readdir*() is declared with the result type *struct dirent* *. (See GA36 in 2.7.3.)

02(C) If *readdir*() is defined as a macro when the header `<dirent.h>` is included:

When the macro *readdir*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *struct dirent* *. (See GA37 in 2.7.3.)

03(C) If *readdir*() is defined as a macro in the header `<dirent.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

5.1.2.2.2 Description (5.1.2.2)

04(A) A call to *readdir*(*dirp*) returns a pointer to a structure representing the entry at the current position in the directory stream.

Testing Requirements:

Test for file names consisting of 1 to {NAME_MAX} bytes in length.

05(A) A call to *readdir*() advances the directory stream to the next directory entry.

06(A) A call to *readdir*() upon reaching the end of the directory stream returns a **NULL** pointer, and *errno* is not changed.

07(B) A call to *readdir*() does not return directory entries containing empty names.

See Reason 3 in Section 5. of POSIX.3 {4}.

D01(C) If it is documented whether the pointer returned by *readdir* () points to static data that may be overwritten by subsequent calls to *readdir*() on the same directory stream:

The details are contained in 5.1.2.2 of the PCD.1.

08(A) The structure representing the directory entry associated with one directory stream is not overwritten by a call to *readdir*() on a different directory stream.

D02(C) If it is documented whether the *readdir*() function buffers several directory entries per actual read operation:

The details are contained in 5.1.2.2 of the PCD.1.

09(A) When a call to *readdir*() is made that causes a read to the underlying directory, then the *st_atime* time-related field of the directory is marked for update.

10(A) When a call to *fork*() is successful and the child process does not continue to process the directory stream, then the parent may continue processing the directory stream using *readdir*() and *rewinddir*() or both.

11(A) When a call to *fork*() is successful and the parent process does not continue to process the directory stream, then the child may continue processing the directory stream using *readdir*() and *rewinddir*() or both.

D03(C) If the action taken when using a directory stream after one of the `exec` type function calls is documented:

The details on its effect are contained in 5.1.2.2 of the PCD.1.

D04(C) If the action taken when both the parent and child processes call *readdir*() after a call to *fork*() is documented:

The details on its effect are contained in 5.1.2.2 of the PCD.1.

5.1.2.2.3 Returns (5.1.2.3)

R01 When a call to *readdir*() completes successfully, then it returns an object of type *struct dirent* *. (See Assertion 1 in 5.1.2.2.1.)

R02 When a call to *readdir*() completes successfully and the end of directory is encountered, then a **NULL** pointer is returned and *errno* is not changed. (See Assertion 6 in 5.1.2.2.1.)

R03 When a call to *readdir*() completes unsuccessfully, then a value of NULL is returned and sets *errno* to indicate the error. (See Assertion 12 in 5.1.2.2.4.)

5.1.2.2.4 Errors (5.1.2.4)

D05(C) If the action associated with passing a *dirp* argument to *readdir*() that does not refer to a currently open directory stream is documented:
The details on its effect are contained in 5.1.2.2 of the PCD.1.

D06(C) If the implementation supports the detection of [EBADF] for *readdir*():
The details under which [EBADF] occurs for *readdir*() are contained in 5.1.2.4 of the PCD.1. (See DGA02 in 2.4.)

12(C) If {PCD_DIR_TYPE} is **TRUE** and the implementation supports the detection of [EBADF] *for readdir*():
When the *dirp* argument does not refer to an open directory stream, then a call to *readdir*(*dirp*) returns a value of NULL and sets *errno* to [EBADF].

13(D) If the implementation does not support the detection of [EBADF] for *readdir*():
A call to *readdir*(*dirp*) is successful (unless a different error condition is detected). (See GA26 in 2.4.)

See Reason 2 in Section 5. of POSIX.3 {4}.

5.1.2.3 *rewinddir*() (5.1.2)

5.1.2.3.1 Synopsis (5.1.2.1)

01(C) If the implementation provides C Standard {2} support:
When the header `<dirent.h>` is included, then the function prototype `void rewinddir(DIR *)` is declared. (See GA36 in 2.7.3.)

D01(C) If the implementation provides Common-Usage C support:
The result type for function *rewinddir*() is contained in 5.1.2.1 of the PCD.1. (See DGA01 in 1.3.3.3.)

02(D) If the implementation provides C Standard {2} support and *rewinddir*() is defined as a macro when the header `<dirent.h>` is included:
When the macro *rewinddir*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *void*. (See GA37 in 2.7.3.)

See Reason 2 in Section 5. of POSIX.3 {4}.

03(C) If *rewinddir*() is defined as a macro in the header `<dirent.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary. (See GA01 in 1.3.3.)

5.1.2.3.2 Description (5.1.2.2)

04(A) A call to *rewinddir*(*dirp*) resets the position of the named directory stream to which *dirp* refers to the beginning of the directory.

05(A) A call to *rewinddir*(*dirp*) causes the directory stream to refer to the current state of the corresponding directory, as a call to *opendir*() would have done.

D02(C) If the action taken when a file is removed from or added to the directory after the most recent call to *rewinddir*() is documented:

The details on whether a subsequent call to *readdir*() returns an entry for that file is contained in 5.1.2.2 of the PCD.1.

D03(C) If the action associated with passing a *dirp* argument to *rewinddir*() that does not refer to a currently open directory stream is documented:

The details on its effect are contained in 5.1.2.2 of the PCD.1.

D04(C) If the action taken when parent and child processes both continue processing of a directory stream with a combination of *readdir*() and *rewinddir*() is documented:

The details on its effect are contained in 5.1.2.2 of the PCD.1.

5.1.2.3.3 Returns (5.1.2.3)

There are no assertions specific to this subclause.

5.1.2.3.4 Errors (5.1.2.4)

There are no assertions specific to this subclause.

5.1.2.4 *closedir*() (5.1.2)

5.1.2.4.1 Synopsis (5.1.2.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<dirent.h>` is included, then the function prototype `int closedir(DIR *)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<dirent.h>` is included, then the function *closedir*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *closedir*() is defined as a macro when the header `<dirent.h>` is included:

When the macro *closedir*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *closedir*() is defined as a macro in the header `<dirent.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

5.1.2.4.2 Description (5.1.2.2)

04(A) When *closedir*() is called on an open directory stream, then it returns a value of zero and makes the directory stream no longer accessible to *readdir*().

05(C) If {PCD_DIR_TYPE} is **TRUE**:

A call to *closedir*() closes the referenced file descriptor.

D01(C) If whether a directory pointer *dirp* continues to point to an accessible object of type DIR after return from a call to *closedir*(*dirp*) is documented:

The details are contained in 5.1.2.2 of the PCD.1.

D02(C) If the action associated with passing a *dirp* argument to *closedir*() that does not refer to a currently open directory stream is documented:

The details on its effect are contained in 5.1.2.2 of the PCD.1.

5.1.2.4.3 Returns (5.1.2.3)

R01 When a call to *closedir*() completes successfully, then a value of (*int*)0 is returned. (See Assertion 4 in 5.1.2.4.2.)

R02 When a call to *closedir*() completes unsuccessfully, then a value of (*int*)-1 is returned and sets *errno* to indicate the error. (See Assertion 6 in 5.1.2.4.4.)

5.1.2.4.4 Errors (5.1.2.4)

D03(C) If the implementation supports the detection of [EBADF] for *closedir*():
The details under which [EBADF] occurs for *closedir*() are contained in 5.1.2.4 of the PCD.1. (See DGA02 in 2.4.)

06(C) If {PCD_DIR_TYPE} is **TRUE** and the implementation supports the detection of [EBADF] for *closedir*():
When the *dirp* argument does not refer to an open directory stream, then a call to *closedir*(*dirp*) returns a value of (*int*)-1 and sets *errno* to [EBADF].

07(D) If the implementation does not support the detection of [EBADF] for *closedir*():
A call to *closedir*(*dirp*) is successful (unless a different error condition is detected). (See GA26 in 2.4.)

See Reason 2 in Section 5. of POSIX.3 {4}.

5.2 Working Directory

5.2.1 *chdir*()

5.2.1.1 Synopsis

01(A) If the implementation provides C Standard {2} support:
When the header `<unistd.h>` is included, then the function prototype `int chdir(const char *)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<unistd.h>` is included, then the function *chdir*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *chdir*() is defined as a macro when the header `<unistd.h>` is included:
When the macro *chdir*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *chdir*() is defined as a macro in the header `<unistd.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

5.2.1.2 Description

04(A) When the first filename component of the *path* argument is ".", and the pathname does not begin with a slash, then *chdir*() resolves the pathname by locating the second filename component (when specified) in the current working directory. (See GA14 in 2.3.6.)

05(A) When the *path* argument points to the string "`/`", then *chdir*() resolves the pathname to the root directory of the process. (See GA15 in 2.3.6.)

06(A) When the *path* argument points to the string "`///`", then *chdir*() resolves the pathname to the root directory of the process. (See GA16 in 2.3.6.)

07(A) When the *path* argument points to a string beginning with a single slash or beginning with three or more slashes, then *chdir*() resolves the pathname by locating the first filename component of the pathname in the root directory of the process. (See GA17 in 2.3.6.)

08(A) When the first filename component of the *path* argument is "..", the pathname does not begin with a slash, and the current working directory is not the root directory of the process, then *chdir*() resolves the pathname by locating the second filename component (when specified) in the parent directory of the current working directory. (See GA18 in 2.3.6.)

09(A) When the *path* argument points to the string "F1/" and `F1` is a directory, then *chdir*() resolves the pathname to `F1`, which is in the current working directory. (See GA19 in 2.3.6.)

10(A) When the argument *path* points to the string "F1//" and `F1` is a directory, then *chdir*() resolves the pathname to `F1`, which is in the current working directory. (See GA20 in 2.3.6.)

11(A) When the *path* argument points to the string "F1/F2", then *chdir*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA21 in 2.3.6.)

12(A) When the pathname argument points to the string "F1/./F2", then *chdir*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA22 in 2.3.6.)

13(A) When the *path* argument points to the string "F1/../F1/F2", then *chdir*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA23 in 2.3.6.)

14(A) When the *path* argument points to the string "F1//F2", then *chdir*() resolves the pathname to the file `F2` in the directory `F1`. which is in the current working directory. (See GA24 in 2.3.6.)

15(C) If {_POSIX_NO_TRUNC} is not supported in the corresponding directory:

When the *path* component is a string of more than {NAME_MAX} bytes in a directory for which {_POSIX_NO_TRUNC} is not supported, then *chdir*() resolves the pathname component by truncating it to {NAME_MAX} bytes. (See GA25 in 2.3.6.)

16(A) On a call to *chdir*(*path*), the pathname *path* supports *filenames* containing any of the following characters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ -

(See GA02 in 2.2.2.32.)

17(A) On a call to *chdir*(*path*), the pathname *path* retains the unique identity of its upper- and lowercase letters. (See GA03 in 2.2.2.32.)

18(A) When *chdir*(*path*) is granted search access to all components of *path*, then the standard file access control mechanism does not cause *chdir*() to indicate that file access is denied.

19(A) A call to *chdir*(*path*) causes the named directory argument path to become the current working directory and returns a zero value.

5.2.1.3 Returns

R01 When a call to *chdir*() completes successfully, then a value of (*int*)0 is returned. (See Assertion 19 in 5.2.1.2.)

R02 When a call to *chdir*() completes unsuccessfully, then a value of (*int*)-1 is returned, sets *errno* to indicate the error, and the current working directory remains unchanged. (See Assertions 20, 21, and 23–27 in 5.2.1.4.)

5.2.1.4 Errors

20(A) When search permission is denied for a component of the path, then a call to *chdir*(*path*) returns a value of (*int*)-1 and sets *errno* to [EACCES] The current directory remains unchanged.

21({NAME_MAX}≤{PCTS_NAME_MAX}C:UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of *path* is longer than the maximum number of bytes in a filename ({NAME_MAX}), then a call to *chdir*(*path*) returns a value of (*int*)-1 and sets *errno* to [ENAMETOOLONG]. The current directory remains unchanged.

22({NAME_MAX}>{PCTS_NAME_MAX}C:UNTESTED)

If the behavior associated with {POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of *path* equals {PCTS_NAME_MAX}, then a call to *chdir*(*path*) succeeds.

23(A) If {PATH_MAX} ≤ {PCTS_PATH MAX}:

When the length of the *path* argument exceeds the maximum number of bytes in a pathname {PATH_MAX}, then a call to *chdir*(*path*) returns a value of (*int*)-1 and sets *errno* to [ENAMETOOLONG]. The current directory remains unchanged.

Otherwise:

When the length of the *path* argument is {PCTS_PATH_MAX}, then a call to *chdir*() is successful.

24(A) When a component of path is not a directory, then a call to *chdir*(*path*) returns a value of (*int*)-1 and sets *errno* to [ENOTDIR]. The current directory remains unchanged.

25(A) When path does not exist, then a call to *chdir*(*path*) returns a value of (*int*)-1 and sets *errno* to [ENOENT]. The current directory remains unchanged.

26(A) When a component of the path prefix of path does not exist, then a call to *chdir*(*path*) returns a value of (*int*)-1 and sets *errno* to [ENOENT]. The current directory remains unchanged.

27(A) When path is an empty string, then a call to *chdir*(*path*) returns a value of (*int*)-1 and sets *errno* to [ENOENT]. The current directory remains unchanged.

5.2.2 *getcwd*()

5.2.2.1 Synopsis

01(A) If the implementation provides C Standard {2} support:

When the header `<unistd.h>` is included, then the function prototype `char * getcwd(char *, size_t)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<unistd.h>` is included, then the function *getcwd*() is declared with the result type *char* *. (See GA36 in 2.7.3.)

02(C) If *getcwd*() is defined as a macro when the header `<unistd.h>` is included:

When the macro *getcwd*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *char* *. (See GA37 in 2.7.3.)

03(C) If *getcwd*() is defined as a macro in the header `<unistd.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

5.2.2.2 Description

04(A) A call to *getcwd*(*buf, size*) copies the absolute pathname of the current working directory into the character array *buf* and returns a pointer to be result.

D01(C) If the behavior of *getcwd*() when *buf* is a **NULL** pointer is documented:

The details are contained in 5.2.2.2 of the PCD.1.

5.2.2.3 Returns

R01 When a call to *getcwd*() completes successfully, then the *buf* argument is returned. (See Assertion 4 in 5.2.2.2.)

R02 When a call to *getcwd*() completes unsuccessfully, then a **NULL** pointer is returned and sets *errno* to indicate the error. (See Assertions 5–8 in 5.2.2.4.)

D02(C) If the contents of the buffer passed to *getcwd*() after an error is documented:
The details are contained in 5.2.2.3 of the PCD.1.

5.2.2.4 Errors

05(A) When the size argument is equal to zero, then a call to *getcwd*(*buf*, *size*) returns a value of NULL and sets *errno* to [EINVAL].

06(A) When the size argument is greater than zero but smaller than the length of the pathname plus 1, then a call to *getcwd*(*buf*, *size*) returns a value of NULL and sets *errno* to [ERANGE].

D03(C) If the implementation supports the detection of [EACCES] for *getcwd*():
The details under which [EACCES] occurs for *getcwd*() are contained in 5.2.2.4 of the PCD.1. (See DGA02 in 2.4.)

07(A) When read permission is denied for a component of the pathname, then a call to *getcwd*(*buf*, *size*) either returns a value of NULL and sets *errno* to [EACCES] or returns the current working directory into the character array *buf*.

08(A) When search permission is denied for a component of the pathname, then a call to *getcwd*(*buf*, *size*) either returns a value of NULL and sets *errno* to [EACCES] or returns the current working directory into the character array *buf*.

5.3 General File Creation

5.3.1 *open*()

5.3.1.1 Synopsis

01(A) If the implementation provides C Standard {2} support:
When the header `<fcntl.h>` is included, then the function prototype
`int open(const char *, int, ...)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<fcntl.h>` is included, then the function *open*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *open*() is defined as a macro when the header `<fcntl.h>` is included:
When the macro *open*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *open*() is defined as a macro in the header `<fcntl.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

5.3.1.2 Description

04(A) When the first filename component of the *path* argument is ".", and the pathname does not begin with a slash, then *open*() resolves the pathname by locating the second filename component (when specified) in the current working directory. (See GA14 in 2.3.6.)

05(A) When the *path* argument points to the string "`/`", then *open*() resolves the pathname to the root directory of the process. (See GA15 in 2.3.6.)

06(A) When the *path* argument points to the string "`///`", then *open*() resolves the pathname to the root directory of the process. (See GA16 in 2.3.6.)

07(A) When the *path* argument points to a string beginning with a single slash or beginning with three or more slashes, then *open*() resolves the pathname by locating the first filename component of the pathname in the root directory of the process. (See GA17 in 2.3.6)

08(A) When the first filename component of the *path* argument is "..", the pathname does not begin with a slash, and the current working directory is not the root directory of the process, then *open*() resolves the pathname by locating the second filename component (when specified) in the parent directory of the current working directory. (See GA18 in 2.3.6.)

09(A) When the *path* argument points to the string "F1/" and `F1` is a directory, then *open*() resolves the pathname to `F1`, which is in the current working directory. (See GA19 in 2.3.6.)

10(A) When the argument *path* points to the string "F1//" and `F1` is a directory, then open() resolves the pathname to `F1`, which is in the current working directory. (See GA20 in 2.3.6.)

11(A) When the *path* argument points to the string "F1/F2", then *open*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA21 in 2.3.6.)

12(A) When the pathname argument points to the string "F1/./F2", then *open*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA22 in 2.3.6.)

13(A) When the *path* argument points to the string "F1/../F1/F2", then *open*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA23 in 2.3.6.)

14(A) When the *path* argument points to the string "F1//F2", then *open*() resolves the pathname to the file `F2` in the directory `F1` which is in the current working directory. (See GA24 in 2.3.6.)

15(C) If {_POSIX_NO_TRUNC} is not supported in the corresponding directory:

When the *path* component is a string of more than {NAME_MAX} bytes in a directory for which {_POSIX_NO_TRUNC} is not supported, then *open*() resolves the pathname component by truncating it to {NAME_MAX} bytes. (See GA25 in 2.3.6.)

16(A) On a call to *open*(*path, oflag*), the pathname *path* supports *filenames* containing any of the following characters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ -

(See GA02 in 2.2.2.32.)

17(A) On a call to *open*(*path, oflag*), the pathname *path* retains the unique identity of its upper- and lowercase letters. (See GA03 in 2.2.2.32.)

18(A) When *open*(*path, O_RDONLY*) is granted read access to *path* and search access to the path prefix of *path*, then the standard file access control mechanism does not cause *open*() to indicate that file access is denied.

D01(C) If the result of a call to *open*() on a FIFO with O_RDWR set is documented:

The details are contained in 5.3.1.2 of the PCD.1.

19(A) When *open*(*path, O_WRONLY*) is granted write access to *path* and search access to the path prefix of *path*, then the standard file access control mechanism does not cause *open*() to indicate that file access is denied.

20(A) When *open*(*path, O_WRONLY | O_CREAT | O_EXCL, mode*) is granted write and search access to the path prefix of *path*, then the standard file access control mechanism does not cause *open*() to indicate that file access is denied.

21(A) A call to *open*() returns a file descriptor that can be used by other I/O functions to refer to the file specified by argument *path*.

22(A) A call to *open*() returns a file descriptor for the named file that is the lowest file descriptor not currently open for that process.

23(B) The file descriptor returned by *open*() refers to an open file description that is not shared with any other process in the system.

See Reason 3 in Section 5. of POSIX.3 {4}.

24(A) The file offset associated with the file descriptor returned by *open*() is initially set to the beginning of the file.

25(A) A call to *open*() returns a file descriptor that has the FD_CLOEXEC flag clear.

26(A) A file descriptor returned by a call to *open*(*path, O_RDONLY*) can be read and cannot be written.

27(A) A file descriptor returned by a call to *open*(*path, O_WRONLY*) can be written and cannot be read.

28(A) A file descriptor returned by a call to *open*(*path, O_RDWR*) can be read and written.

R01 A file descriptor returned by a call to *open*(*path, O_WRONLY|O_APPEND*) or *open*(*path, O_RDWR|O_APPEND*) has the file offset set to end of file prior to each write. (See Assertion 10 in 6.4.2.)

29(A) When the O_TRUNC flag is not set and the file exists, then a call to *open*() does not change the permission bits, ownership, data of the file, *st_atime*, *st_ctime*, or *st_mtime* fields.

30(A) When the argument *path* does not reference an existing file and the O_CREAT flag is set, then a call to *open*() creates the file.

31(A) The user ID of a newly created file is set to the effective user ID of the process.

D02(C) If the conditions under which the group ID of the new file is set to the group ID of the directory in which the file is being created and the conditions under which the group ID of the file is set to the effective group ID of the calling process for a successful call to *open*() with O_CREAT set are documented:

The details are contained in 5.3.1.2 of the PCD.1.

32(A) The group ID of a newly created file is set to either the effective group ID of the process, or the group ID of the directory in which the file is created.

33(A) When the O_CREAT flag is set and the file does not exist, then a call to *open*() sets the file permission bits of the newly created file to the value of mode except for those bits set in the file mode creation mask of the process.

D03(C) If the effect when *open*() is called with O_CREAT set and when bits in the third argument other than the mode bits are set is documented:

The details are contained in 5.3.1.2 of the PCD.1.

R02 When the O_EXCL and O_CREAT flags are set and the file already exists, then a call to *open*() fails. (See Assertion 52 in 5.3.1.4.)

34(B) When more than one process is executing a call to *open*() for the same file with the O_CREAT and O_EXCL flags set, then the check for the existence of the file and the creation of the file is atomic.

See Reason 3 in Section 5. of POSIX.3 {4}.

D04(C) If the action taken when *open*() is called with O_EXCL set and O_CREAT not set is documented:

The details are contained in 5.3.1.2 of the PCD.1.

R03 When the O_NOCTTY flag is set and *path* identifies a terminal device, then a call to *open*() does not cause the terminal device to become the controlling terminal for the process. (See Assertion 4 in 7.1.1.3.)

35(A) When the O_RDONLY and O_NONBLOCK flags are set and the named file is a FIFO, then a call to *open*() returns without delay.

R04 When the O_WRONLY and O_NONBLOCK flags are set, the named file is a FIFO, and no process has the file opened for reading, then a call to *open*() returns an error. (See Assertion 64 in 5.3.1.4.)

36(A) When the O_RDONLY flag is set, the O_NONBLOCK flag is clear, and the named file is a FIFO, then a call to *open*() blocks until a process opens the file for writing.

37(A) When the O_WRONLY flag is set, the O_NONBLOCK flag is clear, and the named file is a FIFO, then a call to *open*() blocks until a process opens the file for reading.

38(D) If the implementation supports block special files with nonblocking I/O:

When the O_NONBLOCK flag is set and the named file is such a block special file, then a call to *open*() returns without waiting for availability of the device.

See Reason 1 in Section 5. of POSIX.3 {4}.

39(D) If the implementation supports character special files with nonblocking I/O:

When the O_NONBLOCK flag is set and the named file is such a character special file, then a call to *open*() returns without waiting for availability of the device.

NOTE — The case of a terminal device file is covered by Assertion 57 in 7.1.2.4.

See Reason 1 in Section 5. of POSIX.3 {4}.

D05(C) If a block special or character special device that supports nonblocking opens is called by *open*() with O_NONBLOCK set, and the behavior of the device after the *open*() call is documented:

The details are contained in 5.3.1.2 of the PCD.1.

40(D) If the implementation supports block special files with nonblocking I/O:

When the O_NONBLOCK flag is clear and the named file is such a block special file, then a call to *open*() blocks until the device is available.

See Reason 1 in Section 5. of POSIX.3 {4}.

41(D) If the implementation supports character special files with nonblocking I/O:

When the O_NONBLOCK flag is clear and the named file is such a character special file, then a call to *open*() blocks until the device is available.

NOTE — The case of a terminal device file is covered by Assertion 56 in 7.1.2.4.

See Reason 1 in Section 5. of POSIX.3 {4}.

D06(C) If the result is documented of a call to *open*(*path*, *oflag*,…) when path is not a block or character file that supports nonblocking *open*()s, and when argument *path* is not a FIFO being opened with O_RDONLY or O_WRONLY set in *oflag*:

The details are contained in 5.3.1.2 of the PCD.1.

42(A) When an existing regular file is successfully opened with flags O_RDWR | O_TRUNC or with flags O_WRONLY | O_TRUNC set, then the file is truncated to zero length, and the mode and owner are unchanged.

43(A) When a call to *open*(*path*, *oflag*) is made with the O_TRUNC flag set in *oflag*, and when *path* refers to a FIFO special file, then the result of the call to *open*() is the same as with the O_TRUNC flag clear.

44(PCTS_GTI_DEVICE?A:UNTESTED)

When a call to *open*(*path*, *oflag*) is made with the O_TRUNC flag set in *oflag*, and when *path* refers to a terminal device file, then the result of the call to *open*() is the same as with the O_TRUNC flag clear.

Testing Requirements:

Test on the input buffer and, when the general terminal interface device of the implementation buffers output, also test on the output buffer.

D07(A) The details of the effect of O_TRUNC on file types other than regular files, FIFO special files, or terminal device files is contained in 5.3.1.2 of the PCD.1.

D08(C) If the effect of calling *open*() with O_TRUNC and O_RDONLY set is documented:
The details are contained in 5.3.1.2 of the PCD.1.

R05 The file descriptor returned by a successful call to *open*() is set to remain open across `exec` function calls. (See Assertion 25 in 5.3.1.2.)

45(A) When O_CREAT is set and the file did not previously exist, then a successful call to *open*() marks for update the *st_atime*, *st_ctime*, and *st_mtime* fields on the created file, and the *st_ctime* and *st_mtime* fields on the parent directory.

46(A) When O_TRUNC is set and the file named by *path* is an existing regular file, then a successful call to *open*() marks for update the *st_ctime* and *st_mtime* fields on the file.

5.3.1.3 Returns

R06 When a call to *open*() completes successfully, then a nonnegative integer that represents the lowest numbered unused file descriptor is returned. (See Assertion 22 in 5.3.1.2.)

R07 When a call to *open*() completes unsuccessfully, then a value of (*int*)-1 is returned and sets *errno* to indicate the error, and no files are created or modified. (See Assertions 47–56 and 58–67 in 5.3.1.4.)

5.3.1.4 Errors

47(A) When search permission is denied for a component of the path prefix of *path*, then a call to *open*(*path*, *oflag*) returns a value of (*int*)-1 and sets *errno* to [EACCES].

48(A) When the file exists and the requested read and/or write permission is denied, then a call to *open*(*path*, *oflag*) returns a value of (*int*)-1 and sets *errno* to [EACCES].

49(A) When the O_CREAT flag is specified, the file does not exist, and write permission is denied for the parent directory of the file to be created, then a call to *open*(*path*, *oflag*) returns a value of (*int*)-1, sets *errno* to [EACCES], and the file is not created.

50(A) When the O_TRUNC flag is specified and write permission is denied for the file, then a call to *open*(*path*, *oflag*) returns a value of (*int*)-1, sets *errno* to [EACCES], and the file is not truncated.

51(B) When O_NONBLOCK is set, the named file is not a FIFO or terminal device file, and the program would be delayed in the open operation, then a call to *open*() returns a value of (*int*)-1 and sets *errno* to [EAGAIN].

See Reason 1 in Section 5. of POSIX.3 {4}.

52(A) When *oflag* argument has O_CREAT and O_EXCL set and the file named by *path* exists, then a call to *open*(*path*, *oflag*) returns a value of (*int*)-1, sets *errno* to [EEXIST], and does not mark for update either the *st_ctime* and *st_mtime* fields of the file and parent directory or the *st_atime* field of the file.

53(A) When the open operation is interrupted by a signal, then a call to *open*(*path*, *oflag*) returns a value of (*int*)-1, sets *errno* to [EINTR], and does not mark for update either the *st_ctime* and *st_mtime* fields of the file and parent directory or the *st_atime* field of the file.

54(A) When the file named by *path* is a directory and *oflag* is O_WRONLY or O_RDWR, then a call to *open*(*path*, *oflag*) returns a value of (*int*)-1, sets *errno* to [EISDIR], and does not mark for update either the *st_ctime* and *st_mtime* fields of the file and parent directory or the *st_atime* field of the file.

55(A) If {OPEN_MAX} ≤ {PCTS_OPEN_MAX}:

When {OPEN_MAX} files have been opened, then a subsequent call to *open*(*path*, *oflag*, *mode*) returns a value of (*int*)-1 and sets *errno* to [EMFILE].

Testing Requirements:

Test for instances

1) When the file does not exist and the O_CREAT flag specifies that the file is not created, and
2) When the file exists and the O_TRUNC flag specifies that the file is not truncated

Otherwise:

{PCTS_OPEN_MAX} files can be opened.

56({NAME_MAX}≤{PCTS_NAME_MAX}?C:UNTESTED

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of *path* is longer than the maximum number of bytes in a filename {NAME_MAX}, then a call to *open*(*path*, *oflag*) returns a value of (*int*)-1 and sets *errno* to [ENAMETOOLONG].

57({NAME_MAX}>{PCTS_NAME_MAX}?C:UNTESTED

If the behavior associated with {_POSIX_NO TRUNC} is supported for the file:

When the length of a pathname component of *path* equals {PCTS_NAME_MAX}, then a call to *open*(*path*, *oflag*) succeeds.

58(A) If {PATH_MAX}≤{PCTS_PATH_MAX}:

When the length of the *path* argument exceeds the maximum number of bytes in a pathname {PATH_MAX}, then a call to *open*(*path*, *oflag*) returns a value of (*int*)-1 and sets *errno* to [ENAMETOOLONG].

Otherwise:

When the length of the path argument is {PCTS_PATH_MAX}, then a call to *open*() is successful.

59(B) When too many files are currently open in the system, then a call to *open*(*path*, *oflag*) returns a value of (*int*)-1 and sets *errno* to [ENFILE].

Testing Requirements:

Test for instances

1) When the file does not exist and the O_CREAT flag specifies that the file is not created, and
2) When the file exists and the O_TRUNC flag specifies that the file is not truncated

See Reason 1 in Section 5. of POSIX.3 {4}.

60(A) When the *oflag* argument does not have O_CREAT set and the file named by *path* does not exist, then a call to *open*(*path*, *oflag*) returns a value of (*int*)-1 and sets *errno* to [ENOENT].

61(A) When the path prefix of *path* does not exist or *path* points to an empty string, then a call to *open*(*path*, *oflag*) returns a value of (*int*)-1 and sets *errno* to [ENOENT].

62(B) When the directory or file system that would contain the new file cannot be extended, then a call to *open*(*path*, *oflag*) returns a value of (*int*)-1, sets *errno* to [ENOSPC], and the file is not created.

See Reason 1 in Section 5. of POSIX.3 {4}.

63(A) When a component of the path prefix of *path* is not a directory, then a call to *open*(*path*, *oflag*) returns a value of (*int*)-1 and sets *errno* to [ENOTDIR].

64(A) When O_WRONLY and O_NONBLOCK are set, the file named by *path* is a FIFO, and no process has the file opened for reading, then a call to *open*(*path*, *oflag*) returns a value of (*int*)-1 and sets *errno* to [ENXIO].

65(C) If the implementation supports a read-only file system:

When the named file resides on a read-only file system and *oflag* has O_WRONLY or O_RDWR, then a call to *open*(*path*, *oflag*) returns a value of(*int*)-1 and sets *errno* to [EROFS].

66(C) If the implementation supports a read-only file system:

When the named file is to reside on a read-only file system, when *oflag* has O_WRONLY or O_RDWR in combination with O_CREAT set, and when the file does not exist, then a call to *open*(*path*, *oflag*) returns a value of (*int*)-1, sets *errno* to [EROFS], does not mark for update the *st_ctime* and *st_mtime* fields of the parent directory, and the file is not created.

67(C) If the implementation supports a read-only file system:

When the named file resides on a read-only file system, when *oflag* has O_TRUNC set, and when the file does exist, then a call to *open*(*path*, *oflag*) returns a value of (*int*)-1, sets *errno* to [EROFS], does not truncate the file, and does not mark for update the *st_atime*, *st_ctime*, and *st_retiree* fields of the file. (See GA13 in 2.3.5.)

5.3.2 *creat*()

5.3.2.1 Synopsis

01(A) If the implementation provides C Standard {2} support:

When the header `<fcntl.h>` is included, then the function prototype `int creat(const char *, mode_t)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<fcntl.h>` is included, then the function *creat*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *creat*() is defined as a macro when the header `<fcntl.h>` is included:

When the macro *creat*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *creat*() is de fined as a macro in the header `<fcntl.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

5.3.2.2 Description

04(A) When the first filename component of the *path* argument is ".", and the pathname does not begin with a slash, then *creat*() resolves the pathname by locating the second filename component (when specified) in the current working directory. (See GA14 in 2.3.6.)

05(B) When the *path* argument points to the string "`/`", then *creat*() resolves the pathname to the root directory of the process. (See GA15 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

06(B) When the *path* argument points to the string "`///`", then *creat*() resolves the pathname to the root directory of the process. (See GA16 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

07(A) When the *path* argument points to a string beginning with a single slash or beginning with three or more slashes, then *creat*() resolves the pathname by locating the first filename component of the pathname in the root directory of the process. (See GA17 in 2.3.6.)

08(A) When the first filename component of the *path* argument is "..", the pathname does not begin with a slash, and the current working directory is not the root directory of the process, then *creat*() resolves the pathname by locating the second filename component (when specified) in the parent directory of the current working directory. (See GA18 in 2.3.6.)

09(B) When the *path* argument points to the string "F1/" and `F1` is a directory, then *creat* () resolves the pathname to F1, which is in the current working directory. (See GA19 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

10(B) When the *path* argument points to the string "F1//" and `F1`, is a directory, then *creat*() resolves the pathname to `F1`, which is in the current working directory. (See GA20 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

11(A) When the *path* argument points to the string "F1/F2", then *creat*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA21 in 2.3.6.)

12(A) When the *path* argument points to the string "F1/./F2", then *creat*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA22 in 2.3.6.)

13(A) When the *path* argument points to the string "F1/../F1/F2", then *creat*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA23 in 2.3.6.)

14(A) When the *path* argument points to the string "F1//F2", then *creat*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA24 in 2.3.6.)

15(C) If {_POSIX_NO_TRUNC} is not supported in the corresponding directory:

When the *path* component is a string of more than {NAME_MAX} bytes in a directory for which {_POSIX_NO_TRUNC} is not supported, then *creat*() resolves the pathname component by truncating it to {NAME_MAX} bytes. (See GA25 in 2.3.6.)

16(A) On a call to *creat*(*path*, *mode*), the pathname *path* supports *filenames* containing any of the following characters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9. _ -

(See GA02 in 2.2.2.32.)

17(A) On a call to *creat*(*path*, *mode*), the pathname *path* retains the unique identity of its upper- and lowercase letters. (See GA03 in 2.2.2.32.)

18(A) When *creat*(*path*, …) is granted write access to the existing file *path* and search access to the path prefix of *path*, then the standard file access control mechanism does not cause *creat*() to indicate that file access is denied.

19(A) When *creat*(*path*, …) is granted write access to the parent directory of the nonexistent file *path* and search access to the path prefix of *path*, then the standard file access control mechanism does not cause *creat*() to indicate that file access is denied.

20(A) A call to *creat*() returns a file descriptor that can be used by other I/O functions to refer to the file specified by argument *path*.

21(A) A call to *creat*() returns a file descriptor for the named file that is the lowest file descriptor not currently open for that process.

22(B) The file descriptor returned by *creat*() is not shared with any other process in the system.

See Reason 3 in Section 5. of POSIX.3 {4}.

23(A) The file offset associated with the file descriptor returned by *creat*() is initially set to the beginning of the file.

24(A) A call to *creat*() returns a file descriptor that has the FD_CLOEXEC flag clear.

25(A) A file descriptor opened by *creat*() can be written and cannot be read.

26(A) When the file exists, then a call to *creat*() does not change any of the following: permission bits, ownership, and *st_atime*.

27(A) When argument path does not reference an existing file, then a call to *creat*() creates the file.

28(A) The user ID of a newly created file is set to the effective user ID of the process.

29(A) The group ID of a newly created file is set to either the effective group ID of the process or the group ID of the directory in which the file is created.

30(A) When the file does not exist, then a call to *creat*() sets the file permission bits of a newly created file to the value of mode except for those set in the file mode creation mask of the process.

31(A) When the named file is a FIFO, then a call to *creat*() blocks until a process opens the file for reading.

32(D) If the implementation supports block special files with nonblocking I/O:

When the named file is a block special file, then a call to *creat*() blocks until the device is available.

See Reason 1 in Section 5. of POSIX.3 {4}.

33(D) If the implementation supports character special files with nonblocking I/O:

When the named file is a character special file, then a call to *creat*() blocks until the device is available.

NOTE — The case of a terminal device file is covered by Assertion 56 in 7.1.2.4.

See Reason 1 in Section 5. of POSIX.3 {4}.

34(A) When an existing regular file is successfully created, then the file is truncated to zero length, and the mode and owner are unchanged.

35(A) When the *path* argument on a call to *creat*(*path*) refers to a FIFO special file, then the result of the call to *creat*() is the same as for the equivalent call to *open*() with the O_TRUNC flag clear or set.

36(PCTS_GTI_DEVICE?A:UNTESTED)

When the *path* argument on a call to *creat*(*path*) refers to a terminal device file, then the result of the call to *creat*() is the same as for the equivalent call to *open*() with the O_TRUNC flag clear or set.

Testing Requirements:

Test on the input buffer and when the general terminal interface device of the implementation buffers output, also test on the output buffer.

R01 The created file descriptor is set to remain open across `exec` function calls. (See Assertion 24 in 5.3.2.2.)

37(A) When the file did not previously exist, then a successful call to *creat*() marks for update the *st_atime*, *st_ctime*, and *st_mtime* fields on the created file, and the *st_ctime* and *st_mtime* fields on the parent directory.

38(A) When the file already exists, then a successful call to *creat*() marks for update the *st_ctime* and the *st_mtime* fields on the file.

5.3.2.3 Returns

R02 When a call to *creat*() completes successfully, then a nonnegative integer that represents the lowest numbered unused file descriptor is returned. (See Assertion 21 in 5.3.2.2.)

R03 When a call to *creat*() completes unsuccessfully, then a value of (*int*)−1 is returned and sets *errno* to indicate the error, and no files are created or are created or modified. (See Assertions 39–45 and 47–53 in 5.3.2.4.)

5.3.2.4 Errors

39(A) When search permission is denied for a component of the path prefix of *path*, then a call to *creat*(*path*, *mode*) returns a value of (*int*)-1 and sets *errno* to [EACCES].

40(A) When the file exists and write permission is denied on the file, then a call to *creat*(*path*, *mode*) returns a value of (*int*)-1, sets *errno* to [EACCES], and does not truncate the file.

41(A) When the file does not exist and write permission is denied for the parent directory of the file to be created, then a call to *creat*(*path*, *mode*) returns a value of (*int*)-1, sets *errno* to [EACCES], and does not create the file.

42(A) When the file creation is interrupted by a signal, then a call to *creat*(*path*, *mode*) returns a value of (*int*)-1, sets *errno*, to [EINTR], and does not mark for update either the *st_ctime* and *st_mtime* fields of the file and parent directory or the *st_atime* field of the file.

43(A) When the file named by path is a directory, then a call to *creat*(*path*, *mode*) returns a value of (*int*)-1, sets *errno* to [EISDIR], and does not mark for update either the *st_ctime* and *st_mtime* fields of the file and parent directory or the *st_atime* field of the file.

44(A) If {OPEN_MAX}≤{PCTS_OPEN_MAX}:

When {OPEN_MAX} files have been opened, then a subsequent call to *creat*(*path*, *mode*) returns a value of (*int*)-1 and sets *errno* to [EMFILE].

Testing Requirements:

Test for instances

1) When the file does not exist, it is not created, and
2) When the file exists, it is not truncated

Otherwise:

{PCTS_OPEN_MAX} files can be opened.

45({NAME_MAX}≤{PCTS_NAME_MAX}?C:UNTESTED

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of *path* is longer than the maximum number of bytes in a filename {NAME_MAX}, then a call to *creat*(*path*, *mode*) returns a value of (*int*)-1 and sets *errno* to [ENAMETOOLONG].

46({NAME_MAX}>{PCTS_NAME_MAX}?C:UNTESTED

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of *path* equals {PCTS_NAME_MAX}, then a call to *creat*(*path*, *mode*) succeeds.

47(A) If {PATH_MAX} ≤ {PCTS_PATH_MAX}:

When the length of the *path* argument exceeds the maximum number of bytes in a pathname {PATH_MAX}, then a call to *creat*(*path*, *mode*) returns a value of (*int*)-1 and sets *errno* to [ENAMETOOLONG].

Otherwise:

When the length of the *path* argument is {PCTS_PATH_MAX}, then a call to *creat*() is successful.

48(B) When too many files are currently open in the system, then a call to *creat*(*path*, *mode*) returns a value of (*int*)-1 and sets *errno* to [ENFILE].

Testing Requirements:

Test for instances

1) When the file does not exist, it is not created, and
2) When the file exists, it is not truncated

See Reason 1 in Section 5. of POSIX.3 {4}.

49(A) When the path prefix of *path* does not exist or *path* points to an empty string, then a call to *creat*(*path*, *mode*) returns a value of (*int*)-1 and sets *errno* to [ENOENT].

50(B) When the directory or file system that would contain the new file cannot be extended, then a call to *creat*(*path*, *mode*) returns a value of (*int*)-1, sets *errno* to [ENOSPC], and does not create the file.

See Reason 1 in Section 5. of POSIX.3 {4}.

51(A) When a component of the path prefix of *path* is not a directory, then a call to *creat*(*path*, *mode*) returns a value of (*int*)-1 and sets *errno* to [ENOTDIR].

52(C) If the implementation supports a read-only file system:

When the file exists and the file named by *path* resides on a read-only file system, then a call to *creat*(*path*, *mode*) returns a value of (*int*)-1, sets *errno* to [EROFS], does not truncate the file, and does not mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the file. (See GA13 in 5.2.3.)

53(C) If the implementation supports a read-only file system:

When the named file is to reside on a read-only file system and the file does not exist, then a call to *creat*(*path*, *mode*) returns a value of (*int*)-1, sets *errno* to [EROFS], does not mark for update the *st_ctime* and *st_mtime* fields of the parent directory, and does not create the file. (See GA13 in 2.3.5.)

5.3.3 *umask*()

5.3.3.1 Synopsis

01(A) If the implementation provides C Standard {2} support:

When the header `<sys/stat.h>` is included, then the function prototype `mode_t umask(mode_t)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<sys/stat.h>` is included, then the function *umask*() is declared with the result type *mode_t*. (See GA36 in 2.7.3.)

02(C) If *umask*() is defined as a macro when the header `<sys/stat.h>` is included:

When the macro *umask*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *mode_t*. (See GA37 in 2.7.3.)

03(C) If *umask*() is defined as a macro in the header `<sys/stat.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

5.3.3.2 Description

04(A) A call to *umask*(*cmask*) sets the file mode creation mask argument of the process to cmask and returns the previous value of the file mode creation mask.

D01(A) The meaning of bits other than file permission bits in the argument to *umask*() is stated in 5.3.3.2 of the PCD.1.

05(A) A call to *umask*(*cmask*) sets the file mode creation mask argument to cmask. Bits set in the file mode creation mask of the process are used to clear the corresponding permission bits in the mode argument supplied by *open*(*path*, *flags*, *mode*), *creat*(*path*, *mode*), *mkdir*(*path*, *mode*), and *mkfifo*(*path*, *mode*).

5.3.3.3 Returns

R01 A call to *umask*() returns the previous value of the file mode creation mask. (See Assertion 4 in 5.3.3.2.)

06(A) When *cmask* is the return value from a previous call to *umask*(), then a call to *umask*(*cmask*) restores the file mode creation mask of the process to the same value as that prior to the previous call to *umask*().

5.3.3.4 Errors

There are no assertions specific to this subclause.

5.3.4 *link*()

5.3.4.1 Synopsis

01(A) If the implementation provides C Standard {2} support:

When the header `<unistd.h>` is included, then the function prototype `int link(const char *, const char *)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<unistd.h>` is included, then the function *link*() is either declared with result type int or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *link*() is defined as a macro when the header `<unistd.h>` is included:

When the macro *link*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *link*() is defined as a macro in the header `<unistd.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

5.3.4.2 Description

04(A) When the first filename component of the existing argument is "", and the pathname does not begin with a slash, then *link*() resolves the pathname by locating the second filename component (when specified) in the current working directory. (See GA14 in 2.3.6.)

05({PCD_LINK_TO_DIRECTORY} is **TRUE**?B:UNTESTED)

When the existing argument points to the string "/", then *link* () resolves the pathname to the root directory of the process. (see GA15 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

06({PCD_LINK_TO_DIRECTORY} is **TRUE**?B:UNTESTED)

When the existing argument points to the string "`///`", then *link*() resolves the pathname to the root directory of the process. (See GA16 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

07(A) When the *existing* argument points to a string beginning with a single slash or beginning with three or more slashes, then *link*() resolves the pathname by locating the first filename component of the pathname in the root directory of the process. (See GA17 in 2.3.6.)

08(A) When the first filename component of the *existing* argument is "..", the pathname does not begin with a slash, and the current working directory is not the root directory of the process, then *link*() resolves the pathname by locating the second filename component (when specified) in the parent directory of the current working directory. (See GA18 in 2.3.6.)

09(PCTS_APP_LINK_DIR?C:UNTESTED)

If {PCD_LINK_TO_DIRECTORY} is **TRUE**:

When the *existing* argument points to the string "F1/" and `F1` is a directory, then *link*() resolves the pathname to `Fl`, which is in the current working directory. (See GA19 in 2.3.6.)

10(PCTS_APP_LINK_DIR?C:UNTESTED)

If {PCD_LINK_TO DIRECTORY} is **TRUE**:

When the *existing* argument points to the string "F1//" and `F1` is a directory, then *link*() resolves the pathname to `F1`, which is in the current working directory. (See GA20 in 2.3.6.)

11(A) When the *existing* argument points to the string “F1/F2”, then *link*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA21 in 2.3.6.)

12(A) When the *existing* argument points to the string “ F1/./F2”, then *link*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA22 in 2.3.6.)

13(A) When the *existing* argument points to the string “ F1/../F1/F2”, then *link*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA23 in 2.3.6.)

14(A) When the *existing* argument points to the string “ F1//F2”, then *link*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA24 in 2.3.6.)

15(C) If {_POSIX_NO_TRUNC} is not supported in the corresponding directory:

When the *existing* component is a string of more than {_NAME_MAX} bytes in a directory for which {_POSIX_NO_TRUNC} is not supported, then *link*() resolves the pathname component by truncating it to {NAME_MAX} bytes. (See GA25 in 2.3.6.)

16(A) On a call to *link*(*existing*, *new*), the pathname *existing* supports *filenames* containing any of the following characters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ -

(See GA02 in 2.2.2.32.)

17(A) On a call to *link*(*existing*, *new*), the pathname *existing* retains the unique identity of its upper- and lowercase letters (See GA03 in 2.2.2.32.)

18(A) When the first filename component of the *new* argument is “.”, and the pathname does not begin with a slash, then *link*() resolves the pathname by locating the second filename component (when specified) in the current working directory. (See GA14 in 2.3.6.)

19(B) When the *new* argument points to the string “`/`”, then *link*() resolves the pathname to the root directory of the process. (See GA15 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

20(B) When the *new* argument points to the string “`///`”, then *link*() resolves the pathname to the root directory of the process. (See GA16 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

21(A) When the *new* argument points to a string beginning with a single slash or beginning with three or more slashes, then *link*() resolves the pathname by locating the first filename component of the pathname in the root directory of the process. (See GA17 in 2.3.6.)

22(A) When the first filename component of the *new* argument is “..”, the pathname does not begin with a slash, and the current working directory is not the root directory of the process, then *link*() resolves the pathname by locating the second filename component (when specified) in the parent directory of the current working directory. (See GA18 in 2.3.6.)

23(C) If {PCD_LINK TO_DIRECTORY} is **TRUE**:

When the *new* argument points to the string “F1/”and `F1` is a directory, then *link*() resolves the pathname to F1, which is in the current working directory. (See GA19 in 2.3.6.)

24(C) If {PCD_LINK_TO DIRECTORY} is **TRUE**:

When the *new* argument points to the string “F1//” and `F1` is a directory, then *link*() resolves the pathname to `F1`, which is in the current working directory. (See GA20 in 2.3.6.)

25(A) When the *new* argument points to the string “F1/F2”, then *link*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA21 in 2.3.6.)

26(A) When the *new* argument points to the string " F1/./F2", then *link*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA22 in 2.3.6.)

27(A) When the *new* argument points to the string "F1/../F1/F2", then *link*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA23 in 2.3.6.)

28(A) When the *new* argument points to the string "F1//F2", then *link*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA24 in 2.3.6.)

29(C) If {_POSIX_NO_TRUNC} is not supported in the corresponding directory:
When the *new* component is a string of more than {NAME_MAX} bytes in a directory for which {_POSIX_NO_TRUNC} is not supported, then *link*() resolves the pathname component by truncating it to {NAME_MAX} bytes. (See GA25 in 2.3.6.)

30(A) On a call to *link*(*existing*, *new*), the pathname *new* supports *filenames* containing any of the following characters:
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ -

(See GA02 in 2.2.2.32.)

31(A) On a call to *link*(*existing*, *new)*, the pathname *new* retains the unique identity of its upper- and lowercase letters. (See GA03 in 2.2.2.32.)

32(A) When *link*(*existing*, *new*) is granted search access to the path prefix of both *existing* and *new* and write permission to the directory that will contain *new*, then the standard file access control mechanism does not cause *link*() to indicate that file access is denied.

33(A) A call to *link*() creates a new link for the existing file such that the new link to the file allows access via *new* to the same data accessible via existing. The call to *link*() returns a value of zero.

34(A) A successful call to *link*(*existing*, *new*) increments by one the link count of the file associated with *existing*.

35(B) When a call to *link*(*existing*, *new*) creates a new link, then the creation of the link and the increment of the link count is atomic.

See Reason 2 in Section 5. of POSIX.3 {4}.

D01(C) If support or nonsupport of *link*() across file systems is documented:
The details are contained in 5.3.4.2 of the PCD.1.

36(A) After a call to *link*(*existing*, *new*), the new link, *new*, refers to the same file as the link *existing*.

R01 An unsuccessful call to *link*(*existing*, *new*) creates no link, and the link count of the file referenced by *existing* remains unchanged. (See Assertion 42 in 5.3.4.4.)

D02(C) If the implementation supports a method to associate the appropriate privileges for linking to directories with a process:
The details are documented in 2.2.2.4 or in 5.3.4.2 of the PCD.1.

D03(C) If support or nonsupport of *link*()on directories is documented:
The details are contained in 5.3.4.2 of the PCD.1.

37(C) If {PCD_LINK TO_DIRECTORY} is **TRUE**:
When the calling process has the appropriate privileges to link to a directory, then a call. to *link*() creates a link to a directory.

38(C) If {PCD_LINK_TO_DIRECTORY} is not documented:
When the file named by *existing* is a directory and the calling process has the appropriate privileges to link to a directory, then a call to *link*(*existing*, *new*) either creates a link to a directory or returns a value of (*int*)-1, sets *errno* to [EPERM], creates no link, and does not change the link count of the file referenced by *existing.*

D04(C) If support or nonsupport for the condition when the calling process requires permission to access the existing file for *link*() to succeed is documented:

The details are contained in 5.3.4.2 of the PCD.1.

R02 When the calling process does not have permission to access the existing file, then a call to *link*(*existing*, *new*) returns a value of (*int*)–1 and sets *errno* to [EACCES]. (See Assertions 41–44 in 5.3.4.4.)

39(A) A call to *link*() marks for update the *st_ctime* field of the file and the *st_ctime* and *st_retiree* field of the parent directory of the newly created link.

40(C) If {PCD_LINK_FILE_SYSTEM} is **TRUE**:

When the link named by *new* and the file named by *existing* are on different file systems, then a call to *link*(*existing*, *new*) is successful.

R03 When the link named by *new* and the file named by *existing* are on different file systems, then a call to *link*(*existing*, *new*) is either successful or returns a value of (*int*)-1, sets *errno* to [EXDEV], creates no link, and does not change the link count of the file referenced by *existing*. (See Assertions 65 and 66 in 5.3.4.4.)

5.3.4.3 Returns

R04 When a call to *link*() completes successfully, then a value of (*int*)0 is returned. (See Assertion 33 in 5.3.4.2.)

R05 When a call to *link*() completes unsuccessfully, then a value of (*int*)-1 is returned and sets *errno* to indicate the error. (See Assertions 41–47, 49, and 51–66 in 5.3.4.4.)

5.3.4.4 Errors

41(A) When search permission is denied for a component of the path prefix of *existing*, then a call to *link*(*existing*, *new*) returns a value of (*int*)-1, sets *errno* to [EACCES], and creates no link.

42(A) When search permission is denied for a component of the path prefix of *new*, then a call to *link*(*existing*, *new*) returns a value of (*int*)-1, sets *errno* to [EACCES], creates no link, and does not change the link count of the file referenced by *existing*.

43(A) When the requested link requires writing in a directory with a mode that denies write permission, then a call to *link* (*existing*, *new*) returns a value of (*int*)-1, sets *errno* to [EACCES], and does not change the link count of the file referenced by *existing*.

44(D) If the implementation requires access permission to link to the file:

When the calling process does not have permission to access the existing file, then a call to *link*(*existing*, *new*) returns a value of (*int*)-1, sets *errno* to [EACCES], and creates no link.

See Reason 2 in Section 5. of POSIX.3 {4}.

45(A) When the link named by *new* exists, then a call to *link*(*existing*, *new*) returns a value of (*int*)-1, sets *errno* to [EEXIST], and does not change the link count of the file referenced by *existing*.

46(A) If {LINK_MAX} ≤ {PCTS_LINK_MAX}:

When {LINK_MAX} links to the file named by the argument *existing* already exist, then a call to *link*(*existing*, *new*) returns a value of (*int*)-1, sets *errno* to [EMLINK], creates no link, and does not change the link count of the file referenced by *existing*.

Otherwise:

{PCTS_LINK_MAX} links can be created to the file specified by argument *existing*.

47({NAME_MAX}≤{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX_NO TRUNC} is supported for the file:

When the length of a pathname component of *existing* is longer than the maximum number of bytes in a filename {NAME_MAX}, then a call to *link* (*existing*, *new*) returns a value of (*int*)-1, sets *errno* to [ENAMETOOLONG], and creates no link.

48({NAME_MAX}>{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of *existing* equals {PCTS_NAME_MAX}, then a call to *link*(*existing*, *new*) succeeds.

49({NAME_MAX}≤{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of *new* is longer than the maximum number of bytes in a filename {NAME_MAX}, then a call to *link*(*existing*, *new*) returns a value of (*int*)-1, sets *errno* to [ENAMETOOLONG], and does not change the link count of the file referenced by *existing*.

50({NAME_MAX}>{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of *new* equals {PCTS_NAME_MAX}, then a call to *link*(*existing*, *new*) succeeds.

51(A) If {PATH_MAX} ≤ {PCTS_PATH_MAX}:

When the length of the *existing* argument exceeds the maximum number of bytes in a pathname {PATH_MAX}, then a call to *link*(*existing*, *new*) returns a value of (*int*)-1, sets *errno* to [ENAMETOOLONG], and creates no link.

Otherwise:

When the length of the path argument is {PCTS_PATH_MAX}, then a call to *link*() is successful.

52(A) If {PATH_MAX} ≤ {PCTS PATH_MAX}:

When the length of the new argument exceeds the maximum number of bytes in a pathname {PATH_MAX}, then a call to *link*(*existing*, *new*) returns a value of (*int*)-1, sets *errno* to [ENAMETOOLONG], and does not change the link count of the file referenced by *existing*.

Otherwise:

When the length of the path argument is {PCTS_PATH_MAX}, then a call to *link*() is successful.

53(A) When a component of the path prefix of *existing* does not exist, then a call to *link*(*existing*, *new*) returns a value of (*int*)-1, sets *errno* to [ENOENT], and creates no link.

54(A) When a component of the path prefix of *new* does not exist, then a call to *link*(*existing*, *new*) returns a value of (*int*)-1, sets *errno* to [ENOENT], and does not change the link count of the file referenced by existing.

55(A) When the file named by *existing* does not exist, then a call to *link*(*existing*, *new*) returns a value of (*int*)-1, sets *errno* to [ENOENT], and creates no link.

56(A) When *existing* points to an empty string, then a call to *link*(*existing*, *new*) returns a value of (*int*)-1, sets *errno* to [ENOENT], and creates no link.

57(A) When *new* points to an empty string, then a call to *link*(*existing*, *new*) returns a value of (*int*)-1, sets *errno* to [ENOENT], and does not change the link count of the file referenced by *existing*.

58(B) When the directory that would contain the link cannot be extended, then a call to *link*(*existing*, *new*) returns a value of (*int*)-1, sets *errno* to [ENOSPC], creates no link, and does not change the link count of the file referenced by *existing*.

See Reason 1 in Section 5. of POSIX.3 {4}.

59(A) When a component of the path prefix of *existing* is not a directory, then a call to *link*(*existing*, *new*) returns a value of (*int*)-1, sets *errno* to [ENOTDIR], and creates no link.

60(A) When a component of the path prefix of *new* is not a directory, then a call to *link*(*existing*, *new*) returns a value of (*int*)-1, sets *errno* to [ENOTDIR], and does not change the link count of the file referenced by *existing*.

61(C) If {PCD_LINK_TO_DIRECTORY} is **TRUE**:

When the file named by *existing* is a directory and the calling process does not have the appropriate privileges to link a directory, then a call to *link* (*existing, new*) returns a value of (*int*)-1, sets *errno* to [EPERM], creates no link, and does not change the link count referenced by *existing*.

62(C) If {PCD_LINK_TO_DIRECTORY} is **FALSE**:
When the file named by *existing* is a directory, then a call to *link*(*existing, new*) returns a value of (*int*)-1, sets *errno* to [EPERM], creates no link, and does not change the link count of the file referenced by *existing.*

63(C) If {PCD_LINK_TO_DIRECTORY} is not documented:
When the file named by *existing* is a directory and the calling process does not have the appropriate privileges to link to a directory, then a call to *link* (*existing, new*) returns a value of (*int*)-1, sets *errno* to [EPERM], creates no link, and does not change the link count of the file referenced by *existing.*

64(C) If the implementation supports a read-only file system:
When the requested link requires writing in a directory on a read-only file system, then a call to *link* (*existing, new*) returns a value of (*int*)-1, sets *errno* to [EROFS], creates no link, does not change the link count of the file referenced by *existing*, and does not mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the parent directory of new. (See GA13 in 5.2.3.)

65(C) If {PCD_LINK_FILE_SYSTEM} is **FALSE**:
When the link named by *new* and the file named by existing are on different file systems, then a call to *link*(*existing, new*) returns a value of (*int*)-1, sets *errno* to [EXDEV], creates no link, and does not change the link count, of the file referenced by existing.

66(C) If {PCD_LINK_FILE_SYSTEM} is not documented:
When the link named by *new* and the file named by existing are on different file systems:, then a call to *link*(*existing, new*) is eithert successful or returns a value of (*int*)-1, sets *errno* to [EXDEV], creates no link, and does not change the link count of the file referenced by existing.

5.4 Special File Creation

5.4.1 *mkdir*()

5.4.1.1 Synopsis

01(A) If the implementation provides C Standard {2} support:
When the header `<sys/stat.h>` is included, then the function prototype `int mkdir(const char *, mode_t)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<sys/stat.h>` is included, then the function *mkdir*() is either declared with result type int or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *mkdir*() is defined as a macro when the header `<sys/stat.h>` is included:
When the macro *mkdir*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type int. (See GA37 in 2.7.3.)

03(C) If *mkdir*() is defined as a macro in the header `<sys/stat.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

5.4.1.2 Description

04(A) When the first filename component of the *path* argument is ".", and the pathname does not begin with a slash, then *mkdir*() resolves the pathname by locating the second filename component (when specified) in the current working directory. (See GA14 in 2.3.6.)

05(B) When the *path* argument points to the string "`/`", then *mkdir*() resolves the pathname to the root directory of the process. (See GA15 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

06(B) When the *path* argument points to the string "`///`", then *mkdir*() resolves the pathname to the root directory of the process. (See GA16 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

07(A) When the *path* argument points to a string beginning with a single slash or beginning with three or more slashes, then *mkdir*() resolves the pathname by locating the first filename component of the pathname in the root directory of the process. (See GA17 in 2.3.6.)

08(A) When the first filename component of the *path* argument is "..", the pathname does not begin with a slash, and the current working directory is not the root directory of the process, then *mkdir*() resolves the pathname by locating the second filename component (when specified) in the parent directory of the current working directory. (See GA18 in 2.3.6.)

09(B) When the *path* argument points to the string "F1/" and `F1` is a directory, then *mkdir*() resolves the pathname to `F1`, which is in the current working directory. (See GA19 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

10(B) When the *path* argument points to the string "F1//" and `F1` is a directory, then *mkdir*() resolves the pathname to `F1`, which is in the current directory. (See GA20 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

11(A) When the *path* argument points to the string "F1/F2 ", then *mkdir*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA21 in 2.3.6.)

12(A) When the *path* argument points to the string " F1/./F2", then *mkdir*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA22 in 2.3.6.)

13(A) When the *path* argument points to the string " F1/../ F1/F2", then *mkdir*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA23 in 2.3.6.)

14(A) When the *path* argument points to the string " F1//F2 ", then *mkdir*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA24 in 2.3.6.)

15(C) If {_POSIX_NO_TRUNC} is not supported in the corresponding directory:

When the *path* component is a string of more than {_NAME_MAX} bytes in a directory for which {_POSIX_NO_TRUNC} is not supported, then *mkdir*() resolves the pathname component by truncating it to {NAME_MAX} bytes. (See GA25 in 2.3.6.)

16(A) On a call to *mkdir*(*path*, *mode*), the pathname *path* supports filenames containing any of the following characters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9. _ -

(See GA02 in 2.2.2.32.)

17(A) On a call to *mkdir*(*path*, *mode*), the pathname *path* retains the unique identity of its upper- and lowercase letters. (See GA03 in 2.2.2.32.)

18(A) When *mkdir*(*path*, …) is granted write access to the parent directory of *path* and search access to the path prefix of *path*, then the standard file access control mechanism does not cause *mkdir*() to indicate that file access is denied.

19(A) A call to *mkdir*(*path*, *mode*) creates a new directory, with file permission bits determined by the *mode* argument and the file creation mask, and returns a value of zero.

D01(A) The details describing the effect on *mkdir*() of bits other than permission bits being set in *mode* are contained in 5.4.1.2 of the PCD.1.

20(A) The owner ID of the newly created directory is set to the effective user ID of the process.

D02(C) If the conditions under which the group ID of the new directory is set to the group ID of the directory in which the directory is being created and the conditions under which the group ID of the directory is set to the effective group ID of the calling process for a successful call to *mkdir*() are documented:

The details are contained in 5.4.1.2 of the PCD.1.

21(A) The group ID of the newly created directory is set to the effective group ID of the process or to the group ID of the directory in which the directory is being created.

22(A) The newly created directory is an empty directory.

23(A) A call to *mkdir*() marks for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the new directory and the *st_ctime* and *st_mtime* fields of the parent directory.

5.4.1.3 Returns

R01 When a call to *mkdir*() completes successfully, then a value of (*int*)0 is returned. (See Assertion 19 in 5.4.1.2.)

R02 When a call to *mkdir*() completes unsuccessfully, then a value of (*int*)-1 is returned and sets *errno* to indicate the error. (See Assertions 24–29 and 31–36 in 5.4.1.4.)

5.4.1.4 Errors

24(A) When search permission is denied for a component of the path prefix of *path*, then a call to *mkdir*(*path*, *mode*) returns a value of (*int*)-1 and sets *errno* to [EACCES].

25(A) When write permission is denied on the parent directory of the directory to be created, then a call to *mkdir*(*path*, *mode*) returns a value of (*int*)-1 and sets *errno* to [EACCES].

26(A) When the file named by *path* exists, then a call to *mkdir*(*path*, *mode*) returns a value of(*int*)-1 and sets *errno* to [EEXIST].

27({LINK_MAX}≤{PCTS_LINK_MAX}?C:UNTESTED)

If creating a directory causes the link count of the directory in which *path* is to be created to be incremented:

When {LINK_MAX} links already exist in the directory in which *path* is to be created, then a call to *mkdir*(*path*, *mode*) returns a value of (*int*)-1, sets *errno* to [EMLINK], and does not create a directory.

Testing Requirements:

Test is to be performed in a directory with has no more than "." and ".." entries.

28({LINK_MAX}>{PCTS_LINK_MAX}?C:UNTESTED)

If creating a directory causes the link count of the directory in which *path* is to be created to be incremented:

{PCTS_LINK_MAX} directories can be created in one directory.

Testing Requirements:

Test is to be performed in a directory with has no more than "." and ".." entries.

29({NAME_MAX}≤{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of *path* is longer than the maximum number of bytes in a filename {NAME_MAX}, then a call to *mkdir*(*path, mode*) returns a value of (*int*)-1 and sets *errno* to [ENAMETOOLONG].

30({NAME_MAX}>{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of *path* equals {PCTS_NAMEMAX}, then a call to *mkdir* (*path, mode*) succeeds.

31(A) If {PATH_MAX} ≤ {PCTS_PATH_MAX}:

When the length of the path argument exceeds the maximum number of bytes in a pathname {PATH_MAX}, then a call to *mkdir*(*path, mode*) returns a value of (*int*)-1 and sets *errno* to [ENAMETOOLONG].

Otherwise:

When the length of the path argument is {PCTS_PATH MAX}, then a call to *mkdir*() is successful.

32(A) When a component of the path prefix of *path* does not exist, then a call to *mkdir(path, mode)* returns a value of (*int*)-1 and sets *errno* to [ENOENT].

33(A) When the *path* argument points to an empty string, then a call to *mkdir(path, mode)* returns a value of (*int*)-1 and sets *errno* to [ENOENT].

34(B) When the file system does not contain enough space to hold the contents of the new directory or to extend the parent directory of the new directory, then a call to *mkdir(path, mode)* returns a value of (*int*)-1, sets *errno* to [ENOSPC], and does not create a directory.

See Reason 1 in Section 5. of POSIX.3 {4}.

35(A) When a component of the path prefix of *path* is not a directory, then a call to *mkdir(path, mode)* returns a value of (*int*)-1 and sets *errno* to [ENOTDIR].

36(C) If the implementation supports a read-only file system:

When the parent directory resides on a read-only file system, then a call to *mkdir(path, mode)* returns a value of (*int*)-1, sets *errno* to [EROFS], does not create a directory, and does not mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the parent directory. (See GA13 in 2.3.5.)

5.4.2 *mkfifo*()

5.4.2.1 Synopsis

01(A) If the implementation provides C Standard {2} support:

When the header `<sys/stat.h>` is included, then the function prototype `int mkfifo(const char *, mode_t)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<sys/stat.h>` is included, then the function *mkfifo()* is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *mkfifo()* is defined as a macro when the header `<sys/stat.h>` is included:

When the macro *mkfifo()* is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *mkfifo()* is defined as a macro in the header `<sys/stat.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

5.4.2.2 Description

04(A) When the first filename component of the *path* argument is ".", and the pathname does not begin with a slash, then *mkfifo()* resolves the pathname by locating the second filename component (when specified) in the current working directory. (See GA14 in 2.3.6.)

05(B) When the *path* argument points to the string "`/`", then *mkfifo()* resolves the pathname to the root directory of the process. (See GA15 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

06(B) When the *path* argument points to the string "`///`", then *mkfifo()* resolves the pathname to the root directory of the process. (See GA16 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

07(A) When the *path* argument points to a string beginning with a single slash or beginning with three or more slashes, then *mkfifo()* resolves the pathname by locating the first filename component of the pathname in the root directory of the process. (See GA17 in 2.3.6.)

08(A) When the first filename component of the *path* argument is "..", the pathname does not begin with a slash, and the current working directory is not the root directory of the process, then *mkfifo()* resolves the pathname by locating the second filename component (when specified) in the parent directory of the current working directory. (See GA18 in 2.3.6.)

09(B) When the *path* argument points to the string "F1/" and `F1` is a directory, then *mkfifo()* resolves the pathname to `F1`, which is in the current working directory. (See GA19 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

10(B) When the *path* argument points to the string "F1//" and `F1` is a directory, then *mkfifo()* resolves the pathname to `F1`, which is in the current working directory. (See GA20 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

11(A) When the *path* argument points to the string "F1/F2", then *mkfifo()* resolves the pathname to the file `F2` in the directory `Fl`, which is in the current working directory. (See GA21 in 2.3.6.)

12(A) When the *path* argument points to the string "F1/./F2", then *mkfifo()* resolves the pathname to the file `F2` in the directory `Fl`, which is in the current working directory. (See GA22 in 2.3.6.)

13(A) When the *path* argument points to the string "F1/../F1/F2", then *mkfifo()* resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA23 in 2.3.6.)

14(A) When the *path* argument points to the string "F1//F2", then *mkfifo()* resolves the pathname to the file `F2` in the directory, `F1`, which is in the current working directory. (See GA24 in 2.3.6.)

15(C) If {_POSIX_NO_TRUNC} is not supported in the corresponding directory:

When the *path* component is a string of more, than {NAME_MAX} bytes in a directory for which {_POSIX_NO_TRUNC} is not supported, then *mkfifo()* resolves the pathname component by truncating it to {NAME_MAX} bytes. (See GA25 in 2.3.6.)

16(A) On a call to *mkfifo(path, mode)*, the pathname *path* supports *filenames* containing any of the following characters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ -

(See GA02 in 2.2.2.32.)

17(A) On a call to *mkfifo*(*path, mode*), the pathname *path* retains the unique identity of its upper- and lowercase letters. (See GA03 in 2.2.2.32.)

18(A) When *mkfifo(path, …)* is granted write access to the parent directory of *path* and search access to the path prefix of *path*, then the standard file access control mechanism does not cause *mkfifo()* to indicate that file access is denied.

19(A) A call to *mkfifo*() creates a new FIFO special file with file permission bits determined by the *mode* argument and the file creation mask, and returns a value of zero.

D01(A) The details describing the effect on *mkfifo*() of bits other than permission bits being set in *mode* are contained in 5.4.2.2 of the PCD.1.

20(A) The owner ID of the newly created FIFO is set to the effective user ID of the process.

D02(C) If the conditions under which the group ID of the new FIFO is set to the group ID of the directory in which the FIFO is being created and the conditions under which the group ID of the FIFO is set to the effective group ID of the calling process for a successful call to *mkfifo()* are documented:

The details are contained in 5.4.2.2 of the PCD.1.

21(A) The group ID of the newly created FIFO is set to the effective group ID of the process, or the group ID of the directory in which the FIFO is being created.

22(A) A call to *mkfifo*() marks for update the *st_atime*, *st_mtime*, and *st_ctime* fields of the FIFO file created and the *st mtime* and *st_ctime* fields of the parent directory of the FIFO file.

5.4.2.3 Returns

R01 When a call to *mkfifo*() completes successfully, then a value of (*int*)0 is returned. (See Assertion 19 in 5.4.2.2.)

R02 When a call to *mkfifo*() completes unsuccessfully, then a value of (*int*)-1 is returned and sets *errno* to indicate the error. (See Assertions 23–26 and 28–33 in 5.4.2.4.)

5.4.2.4 Errors

23(A) When search permission is denied for any component of the path prefix of *path*, then a call to *mkfifo(path, mode)* returns a value of (*int*)-1 and sets *errno* to [EACCES].

24(A) When write permission is denied on the parent directory of the FIFO to be created, then a call to *mkfifo(path, mode)* returns a value of (*int*)-1 and sets *errno* to [EACCES].

25(A) When the file named by *path* exists, then a call to *mkfifo(path, mode)* returns a value of (*int*)-1 and sets *errno* to [EEXIST].

26({NAME_MAX}≤{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of *path* is longer than the maximum number of bytes in a filename {NAME_MAX}, then a call to *mkfifo(path, mode)* returns a value of (*int*)-1, sets *errno* to [ENAMETOOLONG], and does not create a FIFO.

27({NAME_MAX}>{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of *path* equals {PCTS NAME_MAX}, then a call to *mkfifo(path, mode)* succeeds.

28(A) If {PATH_MAX}≤{PCTS_PATH_MAX}:

When the length of the *path* argument exceeds the maximum number of bytes in a pathname {PATH_MAX}, then a call to *mkfifo(path, mode)* returns a value of (*int*)-1, sets *errno* to [ENAMETOOLONG], and does not create a FIFO.

Otherwise:

When the length of the *path* argument is {PCTS_PATH_MAX}, then a call to *mkfifo*() is successful.

29(A) When a component of the path prefix of *path* does not exist, then a call to *mkfifo(path, mode)* returns a value of (*int*)-1 and sets *errno* to [ENOENT].

30(A) When the *path* argument points to an empty string, then a call to *mkfifo(path, mode)* returns a value of (*int*)-1 and sets *errno* to [ENOENT].

31(B) When the directory that would contain the new file cannot be extended or the file system is out of file allocation resources, then a call to *mkfifo(path, mode)* returns a value of (*int*)-1, sets *errno* to [ENOSPC], and does not create a FIFO.

See Reason 1 in Section 5. of POSIX.3 {4}.

32(A) When a component of the path prefix of *path* is not a directory, then a call to *mkfifo(path, mode)* returns a value of (*int*)-1 and sets *errno* to [ENOTDIR].

33(C) If the implementation supports a read-only file system:

When the named file resides on a read-only file system, then a call to *mkfifo(path, mode)* returns a value of (*int*)-1, sets *errno* to [EROFS], does not create a FIFO, and does not mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the parent directory. (See GA13 in 2.3.5.)

5.5 File Removal

5.5.1 *unlink*()

5.5.1.1 Synopsis

01(A) If the implementation provides C Standard {2} support:

When the header `<unistd.h>` is included, then the function prototype `int unlink(const char *)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<unistd.h>` is included, then the function *unlink*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *unlink*() is defined as a macro when the header `<unistd.h>` is included:

When the macro *unlink*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *unlink*() is defined as a macro in the header `<unistd.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

5.5.1.2 Description

04(A) When the first filename component of the *path* argument is ".", and the pathname does not begin with a slash, then *unlink*() resolves the pathname by locating the second filename component (when specified) in the current working directory. (See GA14 in 2.3.6.)

05(B) When the *path* argument points to the string "`/`", then *unlink*() resolves the pathname to the root directory of the process. (See GA15 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

06(B) When the *path* argument points to the string "`///`", then *unlink*() resolves the pathname to the root directory of the process. (See GA16 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

07(A) When the *path* argument points to a string beginning with a single slash or beginning with three or more slashes, then *unlink*() resolves the pathname by locating the first filename component of the pathname in the root directory of the process. (See GA17 in 2.3.6.)

08(A) When the first filename component of the *path* argument is "..", the pathname does not begin with a slash, and the current working directory is not the root directory of the process, then *unlink*() resolves the pathname by locating the second filename component (when specified) in the parent directory of the current working directory. (See GA18 in 2.3.6.)

09(C) If {PCD_LINK_TO_DIRECTORY} is **TRUE**:

When the *path* argument points to the string "F1/" and `F1` is a directory, then *unlink*() resolves the pathname to Fl, which is in the current working directory. (See GA19 in 2.3.6.)

10(C) If {PCD_LINK_TO_DIRECTORY} is **TRUE:**

When the *path* argument points to the string "F1//" and `F1` is a directory, then *unlink*() resolves the pathname to `F1`, which is in the current working directory. (See GA20 in 2.3.6.)

11(A) When the *path* argument points to the string "F1/F2", then *unlink*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA21 in 2.3.6.)

12(A) When the *path* argument points to the string "F1/./F2", then *unlink*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA22 in 2.3.6.)

13(A) When the *path* argument points to the string "F1/../F1/F2", then *unlink*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA23 in 2.3.6.)

14(A) When the *path* argument points to the string "F1//F2", then *unlink*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA24 in 2.3.6.)

15(C) If {_POSIX_NO_TRUNC} is not supported in the corresponding directory:

When the *path* component is a string of more than {NAME_MAX} bytes in a directory for which {_POSIX_NO_TRUNC} is not supported, then *unlink*() resolves the pathname component by truncating it to {NAME_MAX} bytes. (See GA25 in 2.3.6.)

16(A) On a call to *unlink(path)*, the pathname *path* supports *filenames* containing any of the following characters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ -

(See GA02 in 2.2.2.32.)

17(A) On a call to *unlink(path)*, the pathname *path* retains the unique identity of its upper- and lowercase letters. (See GA03 in 2.2.2.32.)

18(A) When *unlink(path)* is granted write access to the parent directory of *path* and search access to the path prefix of path, then the standard file access control mechanism does not cause *unlink*() to indicate that file access is denied.

19(A) A call to *unlink(path)* removes the link named by *path*, decrements the link count of the file referenced by the link, and returns a value of zero.

20(B) When the link count becomes zero and the file is not opened by a process, then the space occupied by the file is freed.

See Reason 1 in Section 5. of POSIX.3 {4}.

21(B) When the link count becomes zero and the file is not, opened by a process, then the file is no longer accessible.

See Reason 3 in Section 5. of POSIX.3 {4}.

22(A) When one or more processes have the file open when the last link is removed, then the link is removed before *unlink*() returns, but the removal of the file contents is postponed until all references to the file have been closed.

D01(C) If the implementation supports a method to associate the appropriate privileges for unlinking directories with a process:

The details are documented in 2.2.2.4 or in 5.5.1.2 of the PCD.1.

D02(C) If the implementation supports *unlink*() on directories, and this is documented:

The details are contained in 5.5.1.2 of the PCD.1.

23(D) If the implementation supports unlinking a directory:

When the calling process has the appropriate privileges to unlink a directory, then a call to *unlink*() can unlink a directory.

See Reason 1 in Section 5. of POSIX.3 {4}.

24(A) A successful call to *unlink*() marks for update the *st_ctime* and *st_mtime* fields of the parent directory.

25(A) When the link count of the file does not become zero, s successful call to *unlink*() marks for update the *st_ctime* field of the file.

26(D) If the implementation supports unlinking a directory and the implementation does not consider unlinking the directory named by the *path* argument an error when it is being used by the system or, another process:

When the calling process has the appropriate privileges to unlink a directory, and when the directory named by the *path* argument is being used by the system or another process. then a call to *unlink(path)* successfully unlinks the directory *path*.

See Reason 2 in Section 5. of POSIX.3 {4}.

5.5.1.3 Returns

R01 When a call to *unlink*() completes successfully, then a value of (*int*)0 is returned. (See Assertion 19 in 5.5.1.2.)

R02 When a call to *unlink*() completes unsuccessfully, then a value of (*int*)−1 is returned and sets *errno* to indicate the error. (See Assertions 27–30 and 32–39 in 5.5.1.4.)

5.5.1.4 Errors

27(A) When search permission is denied for a component of the path prefix of *path*, then a call to *unlink(path)* returns a value of (*int*)-1, sets *errno* to [EACCES], and does not unlink the directory entry.

28(A) When write permission is denied on the directory containing the link to be removed, then a call to *unlink(path)* returns a value of (*int*)-1, sets *errno* to [EACCES], and does not unlink the directory entry.

D03(C) If the implementation supports the error condition [EBUSY] for *unlink*() when the directory named by the path argument cannot be unlinked because it is being used by the system or another process, and this is documented:

The details are contained in 5.5.1.4 of the PCD.1.

29(D) If the implementation supports unlinking a directory and the implementation considers an attempt to unlink a directory chat is being used by the system or another process to be an [EBUSY] error:

When the calling process has the appropriate privileges to unlink a directory, and when the directory named by the *path* argument cannot be unlinked because it is being used by the system or another process, then a call to *unlink(path)* returns a value of (*int*)-1 and sets *errno* to [EBUSY].

See Reason 2 in Section 5. of POSIX.3 {4}.

30({NAME_MAX}≤{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of *path* is longer than the maximum number of bytes in a filename {NAME_MAX}, then a call to *unlink(path)* returns a value of (*int*)-1 and sets *errno* to [ENAMETOOLONG].

31({NAME_MAX}>{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX NO_TRUNC} is supported for the file:

When the length of a pathname component of *path* equals {PCTS_NAME_MAX}, then a call to *unlink(path)* succeeds.

32(A) If {PATH_MAX}≤{PCTS_PATH_MAX}:

When the length of the *path* argument exceeds the maximum number of bytes in a pathname {PATH_MAX}, then a call to *unlink(path)* returns a value of (*int*)-1 and sets *errno* to [ENAMETOOLONG].

Otherwise:

When the length of the path argument is {PCTS_PATH_MAX}, then a call to *unlink*() is successful.

33(A) When a component of the path prefix of *path* does not exist, then a call to *unlink(path)* returns a value of (*int*)-1 and sets *errno* to [ENOENT].

34(A) When the file named by *path* does not exist, then a call to *unlink(path)* returns a value of (*int*)-1 and sets *errno* to {ENOENT}.

35(A) When the *path* argument points to an empty string, then a call to *unlink(path)* returns a value of (*int*)-1 and sets *errno* to {ENOENT}.

36(A) When a component of the path prefix of *path* is not a directory, then a call to *unlink(path)* returns a value of (*int*)-1 and sets *errno* to {ENOTDIR}.

37(C) If the implementation supports the linking and unlinking of directories:

When the calling process does not have the appropriate privileges to unlink a directory and *path* is a directory, then a call to *unlink(path)* returns a value of (*int*)-1, sets *errno* to [EPERM], and does not unlink the directory.

38(D) If the implementation does not support the unlinking of directories:

When *path* refers to a directory, then a call to *unlink(path)* returns a value of (*int*)-1, sets *errno* to {EPERM}, and does not unlink the directory.

See Reason 1 in Section 5 of POSIX.3 {4}.

39(C) If the implementation supports a read-only file system:

When the named file resides on a read-only file system, then a call to *unlink(path)* returns a value of (*int*)-1, sets *errno* to {EROFS}, does not unlink the file entry, and does not mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the parent. directory of the file. (See GA13 in 2.3.5.)

5.5.2 *rmdir*()

5.5.2.1 Synopsis

01(A) If the implementation provides C Standard {2} support:

When the header `<unistd.h>` is included, then the function prototype `int rmdir(const: char *)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<unistd.h>` is included, then the function *rmdir*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *rmdir*() is defined as a macro when the header `<unistd.h>` is included:

When the macro *rmdir*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *rmdir*() is defined as a macro in the header `<unistd.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

5.5.2.2 Description

04(A) When the first filename component of the *path* argument is ".", and the pathname does not begin with a slash, then *rmdir*() resolves the pathname by locating the second filename component (when specified) in the current working directory. (See GA14 in 2.3.6.)

D01(C) If a call to *rmdir(path)* succeeds when the argument path is the root directory or the current directory of any process, and this is documented:
The details are contained in 5.5.2.2 of the PCD.1.

D02(C) If a call to *rmdir(path)* fails and sets *errno* to [EBUSY] when the argument path is the root directory or the current directory of any process, and this is documented:
The details are contained in 5.5.2.2 of the PCD.1.

05(B) When the *path* argument points to the string "`/`", then *rmdir*() resolves the pathname to the root directory of the process. (See GA15 in 2.3.6.)

See Reason 7 in Section 5. of POSIX.3 {4}.

06(B) When the *path* argument points to the string "`///`", then *rmdir*() resolves the pathname to the root directory of the process. (,See GA16 in 2.3.6.)

See Reason 7 in Section 5. of POSIX.3 {4}.

07(A) When the *path* argument points to a string beginning with a single slash or beginning with three or more slashes, then *rmdir*() resolves the pathname by locating the first filename component of the pathname in the root directory of the process. (See GA17 in 2.3.6.)

08(A) When the first filename component of the *path* argument is "..", the pathname does not begin with a slash, and the current working directory is not the root directory of the process, then *rmdir*() resolves the pathname by locating the second filename component (when specified) in the parent directory of the current working directory. (See GA18 in 2.3.6.)

09(A) When the *path* argument points to the string "F1/" and `F1` is a directory, then *rmdir*() resolves the pathname to `F1`, which is in the current working directory. (See GA19 in 2.3.6.)

10(A) When the *path* argument points to the string "F1//" and `F1` is a directory, then *rmdir*() resolves the pathname to `F1`, which is in the current working directory. (See GA20 in 2.3.6.)

11(A) When the *path* argument points to the string "F1/F2", then *rmdir*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA21 in 2.3.6.)

12(A) When the pathname argument points to the string "F1/./F2", then *rmdir*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA22 in 2.3.6.)

13(A) When the *path* argument points to the string "F1/../F1/F2", then *rmdir*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA23 in 2.3.6.)

14(A) When the *path* argument points to the string "F1//F2", then *rmdir*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA24 in 2.3.6.)

15(C) If {_POSIX_NO_TRUNC} is not supported in the corresponding directory:

When the *path* component is a string of more than {NAME_MAX} bytes in a directory for which {_POSIX_NO_TRUNC} is not supported,then *rmdir*() resolves the pathname component by truncating it to {NAME_MAX} bytes. (See GA25 in 2.3.6.)

16(A) On a call to *rmdir*(*path*), the pathname *path* supports *filenames* containing any of the following characters:
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ -

(See GA02 in 2.2.2.32.)

17(A) On a call to *rmdir*(*path*), the pathname *path* retains the unique identity of its upper- and lowercase letters. (See GA03 in 2.2.2.32.)

18(A) When *rmdir*(*path*) is granted write access to the parent directory of *path* and search access to the *path* prefix of path, then the standard file access control mechanism does not cause *rmdir*() to indicate that file access is denied.

19(A) A call to *rmdir*() removes the empty directory named by *path* and returns a value of zero.

20(B) When the link count of the directory being removed becomes zero and no process has the directory open, then the space occupied by the directory is freed and the directory is no longer accessible.

See Reason 1 in Section 5. of POSIX.3 {4}

21(A) When a process has the directory open and a call to *rmdir*() removes the last link to the directory, then when *rmdir*() returns there are no dot and dot-dot entries in the directory, and no new entries may be created in the directory.

22(B) When a process has the directory open when the last link is removed, then the directory is not removed until all references to the directory have been closed.

See Reason 1 in Section 5. of POSIX.3 {4}.

23(A) A successful call to *rmdir*() marks for update the *st_ctime* and *st_mtime* fields of the parent directory.

24(D) If the implementation allows the removal of a directory that is being used by another process:
When the directory named by the *path* argument is being used by another process, then a call to *rmdir*(*path*) removes the directory and returns successfully.

See Reason 2 in Section 5. of POSIX.3 {4}.

5.5.2.3 Returns

R01 When a call to *rmdir*() completes successfully, then a value of (*int*)0 is returned. (See Assertion 19 in 5.5.2.2.)

R02 When a call to *rmdir*() completes unsuccessfully, then a value of (*int*) - 1 is returned and sets *errno* to indicate the error, and the directory is not removed. (See Assertions 25–29 and 31–36 in 5.5.2.4.)

5.5.2.4 Errors

25(A) When search permission is denied for a component of the path prefix of *path*, then a call to *rmdir*(*path*) returns a value of (*int*)-1, sets *errno* to [EACCES], and does not remove the directory.

26(A) When write permission is denied on the parent directory of the directory to be removed, then a call to *rmdir*(*path*) returns a value of (*int*)-1, sets *errno* to [EACCES], and does not remove the directory.

D03(C) If the implementation supports the error condition [EBUSY] for *rmdir*() when the directory named by the path argument cannot be removed because it is being used by another process:
The details are contained in 5.5.2.4 of the PCD.1. (See DGA02 in 2.4.)

27(D) If the implementation does not allow the removal of a directory that is being used by another process:

When the directory named by the *path* argument cannot be removed because it is being used by another process, and when the implementation considers this to be an error, then a call to *rmdir*(*path*) returns a value of (*int*)-1, sets *errno* to [EBUSY], and does not remove the directory.

See Reason 2 in Section 5. of POSIX.3 {4}.

28(A) When the *path* argument names a directory that is not an empty directory, then a call to *rmdir*(*path*) returns a value of (*int*)-1, sets *errno* to [EEXIST] or [ENOTEMPTY], and does not remove the directory.

29({NAME_MAX}≤{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX_NO TRUNC} is supported for the file:

When the length of a pathname component of *path* is longer than the maximum number of bytes in a filename {NAME_MAX}, then a call to *rmdir*(*path*) returns a value of (*int*)-1 and sets *errno* to [ENAMETOOLONG].

30({NAME_MAX}>{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of *path* equals {PCTS_NAME_MAX}, then a call to *rmdir*(*path*) succeeds.

31(A) If {PATH_MAX} ≤ {PCTS_PATH_MAX}

When the length of the *path* argument exceeds the maximum number of bytes in a pathname {PATH_MAX}, then a call to *rmdir*(*path*) returns a value of (*int*)-1, sets *errno* to [ENAMETOOLONG], and does not remove the directory.

Otherwise:

When the length of the *path* argument is [PCTS_PATH_MAX], then a call to *rmdir*() is successful.

32(A) When a component of the path prefix of *path* does not exist, then a call to *rmdir*(*path*) returns a value of(*int*)-1 and sets *errno* to [ENOENT].

33(A) When the *path* argument names a nonexistent directory, then a call to *rmdir*(*path*) returns a value of (*int*)-1 and sets *errno* to [ENOENT].

34(A) When the *path* argument points to an empty string, then a call to *rmdir*(*path*) returns a value of (*int*)-1 and sets *errno* to [ENOENT].

35(A) When a component of *path* is not a directory, then a call to *rmdir*(*path*) returns a value of(*int*)-1 and sets *errno* to [ENOTDIR].

36(C) If the implementation supports a read-only file system:

When the directory entry to be removed resides on a read-only file system, then a call to *rmdir*(*path*) returns a value of (*int*)-1, sets *errno* to [EROFS], does not remove the named directory entry, and does not mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the parent directory of the named directory entry. (See GA13 in 2.3.5.)

5.5.3 *rename*()

5.5.3.1 Synopsis

01(A) If the implementation provides C Standard {2} support:

When the header `<stdio.h>` is included, then the function prototype `int rename(const char *, const char *)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<stdio.h>` is included, then the function *rename*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *rename*() is defined as a macro when the header `<stdio.h>` is included:

When the macro *rename*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *rename*() is defined as a macro in the header `<stdio.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

5.5.3.2 Description

04(A) When the first filename component of the *old* argument is ".", and the pathname does not begin with a slash, then *rename*() resolves the pathname by locating the second filename component (when specified) in the current working directory. (See GA14 in 2.3.6.)

05(B) When the *old* argument points to the string "`/`", then *rename*() resolves the pathname to the root directory of the process. (See GA15 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

06(B) When the *old* argument points to the string "`///`", then *rename*() resolves the pathname to the root directory of the process. (See GA16 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

07(A) When the *old* argument points to a string beginning with a single slash or beginning with three or more slashes, then *rename*() resolves the pathname by locating the first filename component of the pathname in the root directory of the process. (See GA17 in 2.3.6.)

08(A) When the first filename component of the *old* argument is "..", the pathname does not begin with a slash, and the current working directory is not the root directory of the process, then *rename*() resolves the pathname by locating the second filename component (when specified) in the parent directory of the current working directory. (See GA18 in 2.3.6.)

09(A) When the *old* argument points to the string "F1/" and `F1` is a directory, then *rename*() resolves the pathname to `F1`, which is in the current working directory. (See GA19 in 2.3.6.)

10(A) When the *old* argument points to the string "F1//" and `F1` is a directory, then *rename*() resolves the pathname to `F1`, which is in the current working directory. (See GA20 in 2.3.6.)

11(A) When the *old* argument points to the string "F1/F2", then *rename*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA21 in 2.3.6.)

12(A) When the *old* argument points to the string "F1/./F2", then *rename*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA22 in 2.3.6.)

13(A) When the *old* argument points to the string "F1/../F1/F2", then *rename*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA23 in 2.3.6.)

14(A) When the *old* argument points to the string "F1//F2", then *rename*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA24 in 2.3.6.)

15(C) If {_POSIX_NO_TRUNC} is not supported in the corresponding directory:
When the old component is a string of more than {NAME_MAX} bytes in a directory for which {_POSIX_NO_TRUNC} is not supported, then *rename*() resolves the pathname component by truncating it to {NAME_MAX} bytes. (See GA25 in 2.3.6.)

16(A) On a call to *rename*(*old, new*), the pathname *old* supports *filenames* confining any of the following characters:
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ -

(See GA02 in 2.2.2.32.)

17(A) On a call to *rename*(*old, new*), the pathname *old* retains the unique identity of its upper- and lowercase letters. (See GA03 in 2.2.2.32.)

18(A) When the first filename component of the *new* argument is ".", the argument points to a string beginning with a single slash or beginning with three or more slashes, then *rename*() resolves the pathname by locating the second filename component (when specified) in the current working directory. (See GA14 in 2.3.6.)

19(B) When the *new* argument points to the string "`/`", then *rename*() resolves the pathname to the root directory of the process. (See GA15 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

20(B) When the *new* argument points to the string "`///`", then *rename*() resolves the pathname to the root directory of the process. (See GA16 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

21(A) When the *new* argument points to a string beginning with a slash, then *rename*() resolves the pathname by locating the first filename component of the pathname in the root directory of the process. (See GA17 in 2.3.6.)

22(A) When the first filename component of the *new* argument is "..", the pathname does not begin with a slash, and the current working directory is not the root directory of the process, then *rename*() resolves the pathname by locating the second filename component (when specified) in the parent directory of the current working directory. (See GA18 in 2.3.6.)

23(A) When the *new* argument points to the string "F1/" and `F1` is an empty directory, then *rename*() resolves the pathname to `F1`, which is in the current working directory. (See GA19 in 2.3.6.)

24(A) When the *new* argument points to the string "F1//" and `F1` is an empty directory, then *rename*() resolves the pathname to `F1`, which is in the current working directory. (See GA20 in 2.3.6.)

25(A) When the *new* argument points to the string "F1/F2", then *rename*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA21 in 2.3.6.)

26(A) When the *new* argument points to the string "F1/./F2", then *rename*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA22 in 2.3.6.)

27(A) When the *new* argument points to the string "F1/../F1/F2", then *rename*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA23 in 2.3.6.)

28(A) When the *new* argument points to the string "F1//F2", then *rename*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA24 in 2.3.6.)

29(C) If {_POSIX_NO_TRUNC} is not supported in the corresponding directory:

When the *new* component is a string of more than {NAME_MAX} bytes in a directory for which {_POSIX_NO_TRUNC} is not supported, then *rename*() resolves the pathname component by truncating it to {NAME_MAX} bytes. (See GA25 in 2.3.6.)

30(A) On a call to *rename*(*old, new*), the pathname *new* supports *filenames* containing any of the following characters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ -

(See GA02 in 2.2.2.32.)

31(A) On a call to *rename*(*old, new*), the pathname *new* retains the unique identity of its upper- and lowercase letters. (See GA03 in 2.2.2.32.)

32(A) When *rename*(*old, new*) is granted write access to the parent directories of both old and new and search access to the path prefixes of both *old* and *new*, then the standard file access control mechanism does not cause *rename*() to indicate that file access is denied.

33(A) When old and new do not refer to the same existing file, then after a successful call to *rename*(*old, new*) the old pathname no longer exists, and a value of zero is returned.

34(A) When the *old* and the *new* arguments both refer to links to the same existing file, then a call to *rename*(*old, new*) returns successfully without taking any other action.

35(B) When *new* refers to an existing file, then a link named by argument *new* exists throughout the renaming operation and refers either to the file referred to by argument *new* or *old* before the operation began.

See Reason 1 in Section 5. of POSIX.3 {4}.

36(A) When the link named by the *new* argument exists, then a call to *rename*(*old, new*) removes *new* and renames *old* to *new*.

R01 A call to *rename*(*old, new*) requires write access permission for both the directory containing *old* and the directory containing *new*. (See Assertion 45 in 5.5.3.4.)

37(A) When the *old* argument points to the pathname of a directory, and when the directory named by the *new* argument exists and is empty, then a call to *rename*(*old, new*) removes *new* and renames *old* to *new*.

R02 When the *new* argument names an existing directory, then it is required to be an empty directory. (See Assertion 49 in 5.5.3.4.)

38(B) When the link named by the *new* argument exists, no process has the file open, and the link count of the file becomes zero, then a call to *rename*(*old, new*) frees the space occupied by the file.

See Reason 1 in Section 5. of POSIX.3 {4}.

39(A) When one or more processes have the file argument new open and the last link is removed, then a call to *rename*(*old, new*) removes the link before the *rename*() returns, but the removal of the file contents is postponed until all references to the file have been closed.

D01(C) If it is documented whether the implementation requires write access for the directory named by *old* and, if it exists, for the directory named by *new*, for a successful *rename*(*old, new*) call:

The details are contained in 5.5.3.2 of the PCD.1.

40(C) If {PCD_WRITE_PERM_TO_RENAME} is **FALSE**:

When write permission is denied for a directory pointed to by the *old* or *new* arguments, then a call to *rename*(*old, new*) completes successfully.

R03 When write permission is denied for a directory pointed to by the *old* or *new* arguments, then a call to *rename*(*old, new*) either completes successfully or returns a value of (*int*)-1, sets *errno* to [EACCES], and does not change the named files. (See Assertions 46 and 47 in 5.5.3.4.)

41(D) If the implementation supports a call to *rename*(*old, new*) when the directory named by *old* or *new* is being used by the system or another process:

When either *old* or *new* is being used by the system or another process, then a call to *rename*(*old, new*) successfully renames the directory old to new.

See Reason 2 in Section 5. of POSIX.3 {4}.

42(C) If {PCD_LINK_FILE_SYSTEM} is **TRUE**:

When the links named by *old* and *new* are on different file systems, then a call to *rename*(*old, new*) completes successfully.

R04 When the links named by *old* and *new* are on different file systems, then a call to *rename*(*old, new*) is either successful or returns a value of (*int*)*-1*, sets *errno* to [EXDEV], and does not change the named files. (See Assertions 68 and 69 in 5.5.3.4.)

43(A) On a call to *rename*(*old, new*), *rename*() marks for update the *st_ctime* and *st_mtime* fields of the parent directory of each file.

5.5.3.3 Returns

R05 When a call to *rename*() completes successfully, then a value of (*int*)0 is returned. (See Assertion 33 in 5.5.3.2.)

R06 When a call to *rename*() completes unsuccessfully, then a value of (*int*)-1 is returned and sets *errno* to indicate the *errno*, and the named files are not changed. (See Assertions 44–54, 56–61, and 63–68 in 5.5.3.4.)

5.5.3.4 Errors

44(A) When the path prefix of either *old* or *new* denies search permission, then a call to *rename*(*old, new*) returns a value of (*int*)-1, sets *errno* to [EACCES], and does not change the named files.

45(A) When one of the directories containing *old* or *new* denies write permission, then a call to *rename*(*old, new*) returns a value of (*int*)-1, sets *errno* to [EACCES], and does not change the named files.

46(C) If {PCD_WRITE_PERM_TO_RENAME} is **TRUE**:

When write permission is denied for a directory pointed to by the *old* or *new* arguments, then a call to *rename*(*old, new*) returns a value of (*int*)-1, sets *errno* to [EACCES], and does not change the named files.

47(C) If {PCD WRITE_PERM_TO_RENAME} is not documented:

When write permission is denied for a directory pointed to by the *old* or *new* arguments, then a call to *rename*(*old, new*) either completes successfully or returns a value of (*int*)-1, sets *errno* to [EACCES], and does not change the named files.

D02(C) If the implementation supports the error condition [EBUSY] for *rename*(*old, new*) when the directory *old* or *new* cannot be renamed because it is being used by another process:

The details are contained in 5.5.3.4 of the PCD.1.

48(D) If the implementation considers a call to *rename*(*old, new*) when the directory named by *old* or *new* is being used by the system or another process to be an error:

When the directory named by *old* or *new* is being used by the system or another process, then a call to *rename*(*old, new*) returns a value of (*int*)-1, sets *errno* to [EBUSY], and does not change the named files.

See Reason 2 in Section 5. of POSIX.3 {4}.

49(A) When the link named by *new* is a directory containing entries other than dot and dot-dot, then a call to *rename*(*old, new*) returns a value of (*int*)-1, sets *errno* to [EEXIST] or [ENOTEMPTY], and does not change the named files.

50(A) When *new* and *old* name existing directories and the *new* pathname contains a path prefix that refers to old, then a call to *rename*(*old, new*) returns a value of (*int*)-1, sets *errno* to [EINVAL], and does not change the named files.

51(A) When the *new* argument points to a directory and the *old* argument points to a file that is not a directory, then a call to *rename*(*old, new*) returns a value of(*int*)-1, sets *errno* to [EISDIR], and does not change the named files.

52({NAME_MAX}≤{PCTS_NAME_MAX}?C:UNTESTED

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of the *old* argument is longer than the maximum number of bytes in a filename {NAME_MAX}, then a call to *rename*(*old, new*) returns a value of (*int*)-1, sets *errno* to [ENAMETOOLONG], and does not change the existing files.

53({NAME_MAX}≤{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of the *new* argument is longer than the maximum number of bytes in a filename {NAME_MAX}, then a call to *rename*(*old*, *new*) returns a value of (*int*)-1, sets *errno* to [ENAMETOOLONG], and does not change *old.*

54({NAME_MAX}>{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of the *old* argument equals {PCTS_NAME_MAX}, then a call to *rename*(*old*, *new*) succeeds.

55({NAME_MAX}>{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of the *new* argument equals {PCTS_NAME_MAX}, then a call to *rename*(*old*, *new*) succeeds.

56(A) If {PATH_MAX}≤{PCTS_PATH_MAX}:

When the length of the *old* argument exceeds the maximum number of bytes in a pathname {PATHMAX}, then a call to *rename*(*old*, *new*) returns a value of(*int*)-1, sets *errno* to [ENAMETOOLONG], and does not change the existing files.

Otherwise:

When the length of the *old* argument is {PCTS_PATH_MAX}, then a call to *rename*() is successful.

57(A) If {PATH_MAX}≤{PCTS_PATH_MAX}:

When the length of the *new* argument exceeds the maximum number of bytes in a pathname {PATH_MAX}, then a call to *rename*(*old*, *new*) returns a value of (*int*)-1, sets *errno* to [ENAMETOOLONG], and does not change the *old* file.

Otherwise:

When the length of the *new* argument is {PCTS_PATH_MAX}, then a call to *rename*() is successful.

58(A) When a component of one of the path prefixes *old* or *new* does not exist, then a call to *rename*(*old*, *new*) returns a value of (*int*)-1, sets *errno* to [ENOENT], and does not change the existing files.

59(A) When the link named by the *old* argument does not exist, then a call to *rename*(*old*, *new*) returns a value of (*int*)-1, sets *errno* to [ENOENT], and does not change the existing files.

60(A) When either the *old* or *new* argument points to an empty string, then a call to *rename*(*old*, *new*) returns a value of (*int*)–1, sets *errno* to [ENOENT], and does not change the existing files.

61({LINK_MAX}≤{PCTS_LINK_MAX}?C:UNTESTED)

If {PCD_CREAT_LINK_COUNT} is **TRUE:**

When the file named by *old* is a directory and {LINK_MAX} links already exist to the directory in which *new* is to be created, then a call to *rename*(*old*, *new*) returns a value of (*int*)-1 and sets *errno* to [EMLINK].

62({LINK_MAX}≤{PCTS_LINK_MAX}?C:UNTESTED)

If {PCD_CREAT_LINK_COUNT} is **FALSE:**

When the file named by *old* is a directory and {LINK_MAX} links already exist to the directory in which *new* is to be created, then a call to *rename*(*old*, *new*) succeeds.

63({LINK_MAX}≤{PCTS_LINK_MAX}?C:UNTESTED)

If {PCD_CREAT_LINK COUNT} is not documented:

When the file named by *old* is a directory and {LINK_MAX} links already exist to the directory in which *new* is to be created, then a call to *rename*(*old*, *new*) either succeeds or returns a value of (*int*)-1 and sets *errno* to [EMLINK].

64(B) When the directory that would contain *new* cannot be extended, then a call to *rename*(*old*, *new*) returns a value of (*int*)-1, sets *errno* to [ENOSPC], and does not change the named files.

See Reason 1 in Section 5. of POSIX.3 {4}.

65(A) When a component of the path prefix of either *old* or *new* is not a directory, then a call to *rename*(*old*, *new*) returns a value of (*int*)-1, sets *errno* to [ENOTDIR], and does not change the named files.

66(A) When the *old* argument names a directory and the *new* argument names a nondirectory, then a call to *rename*(*old*, *new*) returns a value of (*int*)-1, sets *errno* to [ENOTDIR], and does not change the named files.

67(C) If the implementation supports a read-only file system:
When the requested operation requires writing in a directory on a read-only file system, then a call to *rename*(*old*, *new*) returns a value of (*int*)-1, sets *errno* to [EROFS], does not change the named files, and does not mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the parent directories of the named files. (See GA13 in 5.2.3.)

68(C) If {PCD_LINK_FILE_SYSTEM} is **FALSE**:"
When the links named by *old* and *new* are on different file systems, then a call to *rename*(*old*, *new*) returns a value of (*int*)-1, sets *errno* to [EXDEV], and does not change the named files.

69(C) If {PCD_LINK_FILE_SYSTEM} is not documented:
When the links named by *old* and *new* are on different file systems, then a call to *rename*(*old*, *new*) is either successful or returns a value of (*int*)-1, sets *errno* to [EXDEV], and does not change the named files.

5.6 File Characteristics

5.6.1 File Characteristics: Header and Data Structure

01(A) The following elements are defined in *struct stat* with the types as specified in Table 5.1.

Table 5.1—*stat* Structure

Element	Type
st_mode	mode_t
st_ino	ino_t
st_dev	dev_t
st_nlink	nlink_t
st_uid	uid_t
st_gid	gid_t
st_size	off_t
st_atime	time_t
st_mtime	time_t
st_ctime	time_t

02(A) When the header <sys/stat.h> is included, then the *struct stat* is defined.

GA51 For *stat*() and *fstat*():
The *st_mode* member of the stat structure contains the file mode associated with the file referenced.

GA52 For *stat*() and *fstat*():
The *st_ino* and *st_dev* members of the stat structure taken together provide a unique reference to the file.

GA53 For *stat*() and *fstat*():

The *st_nlink* member of the stat structure contains the number of links associated with the file referenced.

GA54 For *stat*() and *fstat*():

The *st_uid* member of the stat structure contains the user ID associated with the owner of the file referenced.

GA55 For *stat*() and *fstat*():

The *st_gid* member of the stat structure contains the group ID associated with the group owner of the file referenced.

D01(C) If the usage of the field *st_size* in the structure returned by *stat*() and *fstat*() for nonregular files is documented:

The details are contained in 5.6.1 of the PCD.1.

GA56 For *stat*() and *fstat*():

The *st_size* member of the stat structure contains the file size associated with the file referenced.

GA57 For *stat*() and *fstat*():

The *st_atime* member of the stat structure contains the time of last access associated with the file referenced.

GA58 For *stat*() and *fstat*():

The *st_mtime* member of the stat structure contains the time of last data modification associated with the file referenced.

GA59 For *stat*() and *fstat*():

The *st_ctime* member of the stat structure contains the time of last file status change associated with the file referenced.

5.6.1.1 <sys/stat.h> File Types

03(A) When the argument *m* (the *st_mode* member of *struct stat*) refers to a directory file, then the macro *S_ISDIR(m)* evaluates to nonzero. When the argument *m* refers to another file type, then the macro *S_ISDIR(m)* evaluates to zero.

04(C) If the implementation supports character special files:

When the argument *m* (the *st_mode* member of *struct stat*) refers to a character special file, then the macro *S_ISCHR(m)* evaluates to nonzero. When the argument *m* refers to another file type, then the macro *S_ISCHR(m)* evaluates to zero.

05(C) If the implementation supports block special files:

When the argument *m* (the *st_mode* member of *struct stat*) refers to a block special file, then the macro *S_ISBLK*(*m*) evaluates to nonzero. When the argument *m* refers to another file type, then the macro *S_ISBLK*(*m*) evaluates to zero.

06(A) When the argument *m* (the *st_mode* member of *struct star*) refers to a regular file, then the macro *S_ISREG(m)* evaluates to nonzero. When the argument m refers to another file type, then the macro *S_ISREG(m)* evaluates to zero.

07(A) When the argument *m* (the *st_mode* member of *struct star*) refers to a pipe or FIFO file, then the macro *S_ISFIFO(m)* evaluates to nonzero. When the argument *m* refers to another file type, then the macro *S_ISFIFO(m)* evaluates to zero.

5.6.1.2 <sys/stat.h> File Modes

08(A) When the header <sys/stat.h> is included, then the mode value for symbols S_IRWXU, S_IRUSR, S_IWUSR, S_IXUSR, S_IRWXG, S_IRGRP, S_IWGRP, S_IXGRP, S_IRWXO, S_IROTH, S_IWOTH, S_IXOTH, S_ISUID, and S_ISGID are defined.

09(A) When the file mode is specified by the symbol as shown in Table 5.2, the stated permission is provided for the file owner class.

10(A) When the file mode is specified by the symbol as shown in Table 5.3, the stated permission is provided for the file group class.

11(A) When the file mode is specified by the symbol as shown in Table 5.4, the stated permission is provided for the file other class.

Table 5.2—File Owner Class Permission Masks

Symbol	Permission
S_IRUSR	read
S_IWUSR	write
S_IXUSR	execute

Table 5.3—File Group Class Permission Masks

Symbol	Permission
S_IRGRP	read
S_IWGRP	write
S_IXGRP	execute

Table 5.4—File Other Class Permission Masks

Symbol	Permission
S_IROTH	read
S_IWOTH	write
S_IXOTH	execute

12(A) The symbols S_IRWXU, S_IRWXG, and S_IRWXO are masks for read, write, and search (for a directory) or execute (for other file types) permissions for the file owner, group, and other class respectively.

13(A) When the S_ISUID mask is set on a file and the file is executed by one of the exec type functions, then the effective user ID of the process is set to that of the owner of the file.

14(A) When the S_ISGID mask is set on a file and the file is executed by one of the exec type functions, then the effective group ID of the process is set to that of the group owner of the file.

15(A) The values of the symbols S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH, S_IWOTH, S_IXOTH, S_ISUID, and S_ISGID are bitwise discrete.

16(A) The bits defined by the symbol S_IRWXU are the bitwise inclusive OR of S_IRUSR, S_IWUSR, and S_IXUSR.

17(A) The bits defined by the symbol S_IRWXG are the bitwise inclusive OR of S_IRGRP, S_IWGRP, and S_IXGRP.

18(A) The bits defined by the symbol S_IRWXO are the bitwise inclusive OR of S_IROTH, S_IWOTH, and S_IXOTH.

D02(C) If the implementation ORs bits other than those in S_IRWXU, S_IRWXG, and S_IRWXO into the *st_mode* field of the structure returned by *stat*() and *fstat*():

The details are contained in 5.6.1.2 of the PCD.1.

5.6.1.3 <sys/stat.h> Time Entries

19(B) The time-related fields are measured in seconds since the Epoch (00:00:00, January 1, 1970 Coordinated Universal Time).

See Reason 1 in Section 5. of POSIX.3 {4}.

5.6.2 Get File Status

5.6.2.1 *stat*() (5.6.2)

5.6.2.1.1 Synopsis (5.6.2.1)

01(A) If the implementation provides C Standard {2} support:

When the header <sys/stat.h> is included, then the function prototype `int stat(const char *, struct stat *)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header <sys/stat.h> is included, then the function *stat*() is either declared with result type *int* not declared in the header. (See GA36 in 2.7.3.)

02(C) If *stat*() is defined as a macro when the header <sys/stat.h> is included:

When the macro *stat*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *stat*() is defined as a macro in the header <sys/stat.h>:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

5.6.2.1.2 Description (5.6.2.2)

04(A) When the first filename component of the *path* argument is ".", and the pathname does not begin with a slash, then *stat*() resolves the pathname by locating the second filename component (when specified) in the current working directory. (See GA14 in 2.3.6.)

05(A) When the *path* argument points to the string "/", then *stat*() resolves the pathname to the root directory of the process. (See GA15 in 2.3.6.)

06(A) When the *path* argument points to the string "///", then *stat*() resolves the pathname to the root directory of the process. (See GA16 in 2.3.6.)

07(A) When the *path* argument points to a string beginning with a single slash or beginning with three or more slashes, then *stat*() resolves the pathname by locating the first filename component of the pathname in the root directory of the process. (See GA17 in 2.3.6.)

08(A) When the first filename component of the *path* argument is "..", the pathname does not begin with a slash, and the current working directory is not the root directory of the process, then *stat*() resolves the pathname by

locating the second filename component (when specified) in the parent directory of the current working directory. (See GA18 in 2.3.6.)

09(A) When the *path* argument points to the string "F1/" and `F1` is a directory, then *stat*() resolves the pathname to `F1`, which is in the current working directory. (See GA19 in 2.3.6.)

10(A) When the argument *path* points to the string "F1//" and `F1` is a directory, then *stat*() resolves the pathname to `F1`, which is in the current working directory. (See GA20 in 2.3.6.)

11(A) When the *path* argument points to the string "F1/F2", then *stat*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA21 in 2.3.6.)

12(A) When the *path* argument points to the string "F1/./F2", then *stat*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA22 in 2.3.6.)

13(A) When the *path* argument points to the string "F1/../FI,F2", then *stat*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA23 in 2.3.6.)

14(A) When the *path* argument points to the string "F1//F2", then *stat*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA24 in 2.3.6.)

15(C) If {_POSIX_NO_TRUNC} is not supported in the corresponding directory:

When the *path* component is a string of more than {NAME_MAX} bytes in a directory for which {_POSIX_NO_TRUNC} is not supported, then *stat*() resolves the pathname component by truncating it to {NAME_MAX} bytes. (See GA25 in 2.3.6.)

16(A) On a call to *stat*(*path, buf*), the pathname *path* supports *filenames* containing any of the following characters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ -

(See GA02 in 2.2.2.32.)

17(A) On a call to *stat*(*path, buf*), the pathname *path* retains the unique identity of its upper- and lowercase letters. (See GA03 in 2.2.2.32.)

18(A) A successful call to *stat*(*path, buf*) stores, in the *st_mode* member of the *struct stat* pointed to by *buf*, the file mode associated with *path* and returns a value of zero. (See GA51 in 5.6.1.)

19(A) A successful call to *stat*(*path, buf*) stores, in the *st_ino* and *st_dev* members of the *struct stat* pointed to by *buf*, a unique reference to the file—when *st_ino* and *st_dev* are taken together—and returns a value of zero. (See GA52 in 5.6.1.)

20(A) A successful call to *stat*(*path, buf*) stores, in the *st_nlink* member of the *struct stat* pointed to by *buf*, the number of links associated with the file referenced and returns a value of zero. (See GA53 in 5.6.1.)

21(A) A successful call to *stat*(*path, buf*) stores, in the *st_uid* member of the *struct stat* pointed to by *buf*, the user ID associated with the owner of the file referenced and returns a value of zero. (See GA54 in 5.6.1.)

22(A) A successful call to *stat*(*path, buf*) stores, in the *st_gid* member of the *struct stat* pointed to by *buf*, the group ID associated with the group owner of the file referenced and returns a value of zero. (See GA55 in 5.6.1.)

23(A) A successful call to *stat*(*path, buf*) stores, in the *st_size* member of the *struct stat* pointed to by *buf*, the file size associated with the file referenced and returns a value of zero. (See GA56 in 5.6.1.)

24(A) A successful call to *stat*(*path, buf*) stores, in the *st_atime* member of the *struct stat* pointed to by *buf*, the time of last access associated with the file referenced and returns a value of zero. (See GA57 in 5.6.1.)

25(A) A successful call to *stat*(*path, buf*) stores, in the *st_mtime* member of the *struct stat* pointed to by *buf*, the time of last data modification associated with the file referenced and returns a value of zero. (See GA58 in 5.6.1.)

26(A) A successful call to *stat*(*path, buf*) stores, in the *st_ctime* member of the *struct stat* pointed to by *buf*, the time of last file status change associated with the file referenced and returns a value of zero. (See GA59 in 5.6.1.)

27(A) When *stat*(*path, bur*) is granted search access to the path prefix of *path*, then the standard file access control mechanism does not cause *stat*() to indicate that file access is denied.

R01 A successful call to *stat*(*path, buf*) stores a struct stat containing information about the file named by *path* in the area pointed to by *buf* and returns a value of zero. (See Assertions 18–26 in 5.6.2.1.2.)

28(A) A call to *stat*() does not need read, write, or execute permissions to the named file.

D01(C) If additional or alternate file access control mechanisms cause the *stat*() function to fail in ways not specified by POSIX.1 {3}:

The details when additional or alternate file access control mechanisms cause the *stat*() function to fail are contained in 5.6.2.2 of the PCD.1.

29(A) A call to *stat*(*path, buf*) updates any time-related fields marked for update for the file referenced by *path* before writing into the *struct star* and does not update time-related fields not marked for update. (See GA12 in 2.3.5.)

5.6.2.1.3 Returns (5.6.2.3)

R02 When a call to *stat*() completes successfully, then a value of (*int*)0 is returned. (See Assertions 18–26 in 5.6.2.1.2.)

R03 When a call to *stat*() completes unsuccessfully, then a value of (*int*)-1 is returned and sets *errno* to indicate the error. (See Assertions 30, 31 and 33–37 in 5.6.2.1.4.)

5.6.2.1.4 Errors (5.6.2.4)

30(A) When search permission is denied for a component of the path prefix of *path*, then a call to *stat*(*path, bud*) returns a value of (*int*)-1 and sets *errno* to [EACCES].

31({NAME_MAX}≤{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of *path* is longer than the maximum number of bytes in a filename {NAME_MAX}, then a call to *stat*(*path, buf*) returns a value of (*int*)-1 and sets *errno* to [ENAMETOOLONG].

32({NAME_MAX}>{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of *path* equals {PCTS_NAME_MAX}, then a call to *stat*(*path, buf*) succeeds.

33(A) If {PATH_MAX} ≤ {PCTS_PATH_MAX}:

When the length of the path argument exceeds the maximum number of bytes in a pathname {PATH_MAX}, then a call to *stat*(*path, bud*) returns a value of (*int*)-1 and sets *errno* to [ENAMETOOLONG].

Otherwise:

When the length of the *path* argument is {PCTS_PATH_MAX}, then a call to *stat*() is successful.

34(A) When the file named by *path* does not exist, then a call to *stat*(*path, buf*) returns a value of (*int*)-1 and sets *errno* to [ENOENT].

35(A) When a component of the path prefix of *path* does not exist, then a call to *stat*(*path, buf*) returns a value of (*int*)-1 and sets *errno* to [ENOENT].

36(A) When the *path* argument points to an empty string, then a call to *stat*(*path, buf*) returns a value of (*int*)-1 and sets *errno* to [ENOENT].

37(A) When the component of the path prefix of *path* is not a directory, then a call to *stat*(*path, bud*) returns a value of (*int*)-1 and sets *errno* to [ENOTDIR].

5.6.2.2 *fstat*() (5.6.2)

5.6.2.2.1 Synopsis (5.6.2.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<sys/stat.h>` is included, then the function prototype `int fstat(int, struct stat *)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<sys/stat.h>` is included, then the function *fstat*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *fstat*() is defined as a macro when the header `<sys/stat.h>` is included:

When the macro *fstat*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int.* (See GA37 in 2.7.3.)

03(C) If *fstat*() is defined as a macro in the header `<sys/stat.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

5.6.2.2.2 Description (5.6.2.2)

04(A) A successful call to *fstat*(*fildes, buf*) stores, in the *st_mode* member of the *struct stat* pointed to by *buf,* the file mode associated with *fildes* and returns a value of zero. (See GA51 in 5.6.1.)

05(A) A successful call to *fstat*(*fildes, buf*) stores, in the *st_ino* and *st_dev* members of the *struct stat* pointed to by *buf,* a unique reference to the file—when *st_ino* and *st_dev* are taken together—and returns a value of zero. (See GA52 in 5.6.1.)

06(A) A successful call to *fstat*(*fildes, buf*) stores, in the *st_nlink* member of the *struct stat* pointed to by *buf,* the number of links associated with the file referenced and returns a value of zero. (See GA53 in 5.6.1.)

07(A) A successful call to *fstat*(*fildes, buf*) stores, in the *st_uid* member of the *struct stat* pointed to by *buf,* the user ID associated with the owner of the file referenced and returns a value of zero. (See GA54 in 5.6.1.)

08(A) A successful call to *fstat*(*fildes, buf*) stores, in the *st_gid* member of the *struct stat* pointed to by *buf,* the group ID associated with the group owner of the file referenced and returns a value of zero. (See GA55 in 5.6.1.)

09(A) A successful call to *fstat*(*fildes, buf*) stores, in the *st_size* member of the *struct stat* pointed to by *buf,* the file size associated with the file referenced and returns a value of zero. (See GA56 in 5.6.1.)

10(A) A successful call to *fstat*(*fildes, buf*) stores, in the *st_atime* member of the *struct stat* pointed to by *buf,* the time of last access associated with the file referenced and returns a value of zero. (See GA57 in 5.6.1.)

11(A) A successful call to *fstat*(*fildes, buf*) stores, in the *st_mtime* member of the *struct stat* pointed to by *buf,* the time of last data modification associated with the file referenced and returns a value of zero. (See GA58 in 5.6.1.)

12(A) A successful call to *fstat*(*fildes, buf*) stores, in the *st_ctime* member of the *struct stat* pointed to by *buf,* the time of last file status change associated with the file referenced and returns a value of zero. (See GA59 in 5.6.1.)

R01 A call to *fstat*(*fildes, buf*) stores a *struct stat* containing information about the file associated with the named file descriptor in the area pointed to by *buf* and returns a value of zero. (See Assertions 4–12 in 5.6.2.2.2.)

D01(C) If additional or alternate file access control mechanisms cause the *fstat*() function to fail in ways not specified by POSIX.1 {3}:

The details when additional or alternate file access control mechanisms cause the *fstat*() function to fail are contained in 5.6.2.2 of the PCD.1.

13(A) A call to *fstat*(*fildes, buf*) updates any time-related fields marked for update for the file referenced by *fildes* before writing into the *struct stat* and does not update time-related fields not marked for update. (See GA12 in 2.3.5.)

5.6.2.2.3 Returns (5.6.2.3)

R02 When a call to *fstat*() completes successfully, then a value of (*int*)0 is returned. (See Assertions 4–12 in 5.6.2.2.2.)

R03 When a call to *fstat*() completes unsuccessfully, then a value of (*int*)–1 is returned and sets *errno* to indicate the error. (See Assertion 14 in 5.6.2.2.4.)

5.6.2.2.4 Errors (5.6.2.4)

14(A) When the file descriptor argument *fildes* is not valid, then a call to *fstat*(*fildes, buf*) returns a value of (*int*)-1 and sets *errno* to [EBADF].

5.6.3 *access*()

5.6.3.1 Synopsis

01(A) If the implementation provides C Standard {2} support:

When the header `<unistd.h>` is included, then the function type `int access(const char *, int)` is declared. (See GA36 inproto 2.7.3.)

Otherwise:

When the header `<unistd.h>` is included, then the function *access*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *access*() is defined as a macro when the header `<unistd.h>` is included:

When the macro *access*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *access*() is defined as a macro in the header `<unistd.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

5.6.3.2 Description

04(A) When the first filename component of the *path* argument is ".", and the pathname does not begin with a slash, then *access*() resolves the pathname by locating the second filename component (when specified) in the current working directory. (See GA14 in 2.3.6.)

05(A) When the *path* argument points to the string "`/`", then *access*() resolves the pathname to the root directory of the process. (See GA15 in 2.3.6.)

06(A) When the *path* argument points to the string "`///`", then *access*() resolves the pathname to the root directory of the process. (See GA16 in 2.3.6.)

07(A) When the *path* argument points to a string beginning with a single slash or beginning with three or more slashes, then *access*() resolves the pathname by locating the first filename component of the pathname in the root directory of the process. (See GA17 in 2.3.6.)

08(A) When the first filename component of the *path* argument is "..", the pathname does not begin with a slash, and the current working directory is not the root directory of the process, then *access*() resolves the pathname by locating the second filename component (when specified) in the parent directory of the current working directory. (See GA18 in 2.3.6.)

09(A) When the *path* argument points to the string "F1/" and `F1` is a directory, then *access*() resolves the pathname to `F1`, which is in the current working directory. (See GA19 in 2.3.6.)

10(A) When the argument *path* points to the string "F1//" and F1 is a directory, then *access*() resolves the pathname to `F1`, which is in the current working directory. (See GA20 in 2.3.6.)

11(A) When the *path* argument points to the string "F1/F2", then *access*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA21 in 2.3.6.)

12(A) When the *path* argument points to the string "F1/./F2", then *access*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA22 in 2.3.6.)

13(A) When the *path* argument points to the string "F1/../F1/F2", then *access*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA23 in 2.3.6.)

14(A) When the *path* argument points to the string "F1//F2", then *access*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA24 in 2.3.6.)

15(C) If {_POSIX_NO_TRUNC} is not supported in the corresponding directory:

When the *path* component is a string of more than {NAME_MAX} bytes in a directory for which {_POSIX_NO_TRUNC} is not supported, then *access*() resolves the pathname component by truncating it to {NAMEMAX} bytes. (See GA25 in 2.3.6.)

16(A) On a call to *access(path, amode)*, the pathname *path* supports *filenames* containing any of the following characters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ -

(See GA02 in 2.2.2.32.)

17(A) On a call to *access(path, amode)*, the pathname *path* retains the unique identity of its upper- and lowercase letters. (See GA03 in 2.2.2.32.)

18(A) When the header `<unistd.h>` is included, the symbolic constants R_OK, W_OK, X_OK, and F_OK are defined.

19(A) When the file referenced by *path* exists, then a call to *access(path, amode)* passes tests for *amode* F_OK and returns a value of zero.

20(A) When the real user ID of the calling process matches the owner of the file referenced by *path* and permission (see Table 5.5) is granted for the owner, then a call to *access(path, amode)* succeeds for amode (see Table 5.5) and returns a value of zero.

Table 5.5—Access via File Owner Class

Permission	**amode**
Read	R_OK
Write	W_OK
Execute	X_OK

21(A) When the real user ID of the calling process does not match the owner of the file referenced by *path*, when the real group ID or one of the supplementary group IDs of the calling process matches the group owner of the

file, and when permission (see Table 5.6) is granted for the group, then a call to *access(path, amode)* succeeds for *amode* (see Table 5.6) and returns a value of zero.

Table 5.6—Access via File Group Class

Permission	amode
Read	R_OK
Write	W_OK
Execute	X_OK

22(A) When the real user ID of the calling process does not match the owner of the file referenced by *path*, when neither the real group ID nor any of the supplementary group IDs of the calling process match the group owner of the file, and when permission (see Table 5.7) is granted for others, then a call to *access(path, amode)* succeeds for amode (see Table 5.7) and returns a value of zero.

Table 5.7—Access via File Other Class

Permission	amode
Read	R_OK
Write	W_OK
Execute	X_OK

D01(C) If the implementation provides a mechanism to assign to a process the appropriate privileges to override the file access control mechanism:

The details are contained in 2.2.2.4 or 5.6.3.2 of the PCD.1.

D02(C) If the implementation provides a mechanism to assign to a process the appropriate privileges to override the file access control mechanism and it is documented whether the implementation supports a successful return from *access*(*path, X_OK*) for a process with appropriate privileges when none of the execute bits for *path* are set:

The details are contained in 5.6.3.2 of the PCD 1.

23(C) If the implementation provides a method for associating with a process the appropriate privilege to override the file access control mechanism:

When the process has the appropriate privileges to override the file access control mechanism and the file exists, then a call to *access(path, amode)* will succeed when *amode* is set to F_OK, R_OK, or W_OK.

Testing Requirements:

Test for *amode* set to F_OK, R_OK, and W_OK.

24(C) If the implementation provides a method for associating with a process the appropriate privilege to override the file access control mechanism:

When the process has the appropriate privileges to override the file access control mechanism, then a call to *access(path, amode)* will succeed when *amode* is set to X_OK and any of the execute bits are set or when the file is a directory.

25(A) When *amode* is a bitwise inclusive OR of two or more of R_OK, W_OK, and X_OK, and when separate calls with each of the indicated permissions requested independently would all return successfully, then a call to *access(path, amode)* returns a value of zero.

5.6.3.3 Returns

R01 When a call to *access*() completes successfully, then a value of (*int*)0 is returned. (See Assertions 19–25 in 5.6.3.2.)

R02 When a call to *access*() completes unsuccessfully, then a value of (*int*)-1 is returned and sets *errno* to indicate the error. (See Assertions 26–28 and 30–37 in 5.6.3.4.)

5.6.3.4 Errors

26(A) When any of the requested permissions specified by *amode* are denied, then a call to *access(path, amode)* returns a value of (*int*)-1 and sets *errno* to [EACCES].

27(A) When search permission is denied for a component of the path prefix of *path,* then a call to *access(path, amode)* returns a value of (*int*)-1 and sets *errno* to [EACCES].

28({NAME_MAX}≤{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of *path* is longer than the maximum number of bytes in a filename {NAME_MAX}, then a call to *access(path, amode)* returns a value of (*int*)-1 and sets *errno* to [EACCES].

29({NAME_MAX}>{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of *path* equals {PCTS_NAME_MAX}, then a call to *access(path, amode)* succeeds.

30(A) If {PATH_MAX} ≤ {PCTS_PATH_MAX}:

When the length of the *path* argument exceeds the maximum number of bytes in a pathname {PATH_MAX}, then a call to *access(path, amode)* returns a value of (*int*)-1 and sets *errno* to [ENAMETOOLONG].

Otherwise:

When the length of the path argument is {PCTS_PATH_MAX}, then a call to *access*() is successful.

31(A) When the *path* argument points to an empty string, then a call to *access(path, amode)* returns a value of (*int*)-1 and sets *errno* to [ENOENT].

32(A) When a component of the path prefix of *path* does not exist, then a call to *access(path, arnode)* returns a value of (*int*)-1 anti sets *errno* to [ENOENT].

33(A) When the *path* argument points to name of file that does not exist, then a call to *access(path, amode)* returns a value of (*int*)-1 and sets *errno* to [ENOENT].

34(A) When a component of the path prefix of *path* is not a directory, then a call to *access(path, amode)* returns a value of (*int*)-1 and sets *errno* to [ENOTDIR].

35(C) If the implementation supports a read-only file system:

When write access is requested for a file on a read-only file system, then a call to *access(path, amode)* returns a value, of (*int*)-1, sets *errno* to [EROFS], and does not mark for update the *st_atime,* *st_ctime,* and *st_mtime* fields of the file. (See GA13 in 2.3.5.)

D03(C) If the implementation supports the detection of [EINVAL] for *access*():

The details under which [EINVAL] occurs for *access*() are contained in 5.6.3.4 of the PCD.1.

36(C) If the implementation supports the detection of [EINVAL] for *access*():

When the amode argument is invalid, then a call to *access(path, amode)* returns a value of (*int*)-1 and sets *errno* to [EINVAL].

37(D) If the implementation does not support the detection of [EINVAL] for *access*():

A call to *access(path, amode)* is successful (unless a different error condition is detected). (See GA26 in 2.4.)

See Reason 2 in Section 5. of POSIX.3 {4}.

5.6.4 *chmod*()

5.6.4.1 Synopsis

01(A) If the implementation provides C Standard {2} support:

When the header `<sys/stat.h>` is included, then the function prototype `int chmod(const char *, mode_t)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<sys/stat.h>` is included, then the function *chmod*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *chmod*() is defined as a macro when the header `<sys/stat.h>` is included:

When the macro *chmod*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *chmod*() is defined as a macro in the header `<sys/stat.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

5.6.4.2 Description

04(A) When the first filename component of the *path* argument is ".", and the pathname does not begin with a slash, then *chmod()* resolves the pathname by locating the second filename component (when specified) in the current working directory. (See GA14 in 2.3.6.)

05(PCTS APP_MODE && PCTS_ROOT_WRITABLE?A:UNTESTED)

When the *path* argument points to the string "`/`", then *chmod*() resolves the pathname to the root directory of the process. (See GA15 in 2.3.6.)

06(PCTS APP_MODE && PCTS_ROOT_WRITABLE?A:UNTESTED)

When the *path* argument points to the string "`///`". then *chmod*() resolves the pathname to the root directory of the process. (See GA16 in 2.3.6.)

07(A) When the *path* argument points to a string beginning with a single slash or beginning with three or more slashes, then *chmod*() resolves the pathname by locating the first filename component of the pathname in the root directory of the process. (See GA17 in 2.3.6.)

08(A) When the first filename component of the *path* argument is "..", the pathname does not begin with a slash, and the current working directory is not the root directory of the process, then *chmod*() resolves the pathname by locating the second filename component (when specified) in the parent directory of the current working directory. (See GA18 in 2.3.6.)

09(A) When the *path* argument points to the string "F1/" and `F1` is a directory, then chmod() resolves the pathname to `F1`, which is in the current working directory. (See GA19 in 2.3.6.)

10(A) When the argument *path* points to the string "F1//" and F1 is a directory, then *chmod*() resolves the pathname to `F1`, which is in the current working directory. (See GA20 in 2.3.6.)

11(A) When the *path* argument points to the string "F1/F2", then *chmod*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA21 in 2.3.6.)

12(A) When the *path* argument points to the string “F1/./F2”, then *chmod*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA22 in 2.3.6.)

13(A) When the *path* argument points to the string “F1/../F1/F2”, then *chmod*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA23 in 2.3.6.)

14(A) When the *path* argument points to the string “F1//F2”, then *chmod*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA24 in 2.3.6.)

15(C) If {_POSIX_NO_TRUNC} is not supported in the corresponding directory:

When the *path* component is a string of more than {NAME_MAX} bytes in a directory for which LPOSIX_NO_TRUNC} is not supported,then *chmod*() resolves the pathname component by truncating it to {NAME_MAX} bytes. (See GA25 in 2.3.6.)

16(A) On a call to *chmod*(*path, mode*), the pathname *path* supports *filenames* containing any of the following characters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ -

(See GA02 in 2.2.2.32.)

17(A) On a call to *chmod(path, mode),* the pathname *path* retains the unique identity of its upper- and lowercase letters. (See GA03 in 2.2.2.32.)

18(A) When search access is granted to each component of the path prefix of *path*, then the standard file access control mechanism does not cause *chmod*() to indicate that file access is denied.

19(A) When the effective user ID matches the user ID of the file, then a call to *chmod(path, mode)* sets the file permission bits of the file named by *path* according to the bit pattern contained in the argument *mode* and returns a value of zero.

D01(C) If the implementation provides a mechanism to assign to a process the appropriate privileges to change the file mode:

The details are contained in 2.2.2.4 or 5.6.4.2 of the PCD.1.

20(C) If the implementation provides the mechanism for creating a process with the appropriate privilege to change the file mode:

When the process has the appropriate privileges to change the file mode, then a call to *chmod(path, mode)* sets the file permission bits of the file named by *path* according to the bit pattern contained in the argument *mode* and returns a value of zero

21(PCTS_CHMOD_SET_IDS?A:UNTESTED)

When the effective user ID matches the user ID of the file, then a call to *chmod(path, mode)* sets S_ISUID, S_ISGID, and the file permission bits of the file named by *path* according to the bit pattern contained in the argument *mode* and returns a value of zero.

22(PCTS_CHMOD_SET_IDS?C:UNTESTED)

If the implementation provides a method for associating with a process the appropriate privileges to change the file mode:

When the process has the appropriate privileges to change the file mode, then a call to *chmod(path, mode)* sets S_ISUID, S_ISGID, and the file permission bits of the file named by *path* according to the bit pattern contained in the argument *mode* and returns a value of zero.

D02(C) If restrictions not specified by POSIX.1 {3} cause the S_ISUID or S_ISGID bits to be ignored:

The details when additional restrictions cause the S_ISUID and S_ISGID bits in mode to be ignored are contained in 5.6.4.2 of the PCD.1.

23(PCTS_CHMOD_SET_IDS?A:UNTESTED)

When the calling process does not have the appropriate privileges to change the file mode, and when the group owner of the regular file specified by *path* does not match the effective group ID or one of the supplementary group IDs, then a call to *chmod(path, mode)* clears the S_ISGID bit upon successful completion.

D03(A) The effect on file descriptors for files open at the time of the *chmod*() function are contained in 5.6.4.2 of the PCD.1.

24(A) A successful call to *chmod(path, mode)* marks for update the *st_ctime* field of the file.

5.6.4.3 Returns

R01 When a call to *chmod*() completes successfully, then a value of (*int*)0 is returned. (See Assertions 19–22 in 5.6.4.2.)

R02 When a call to *chmod(path, mode)* completes unsuccessfully, then a value of (*int*)-1 is returned, sets *errno* to indicate the error, and no change is made to the file *mode.* (See Assertions 25, 26, and 28–34 in 5.6.4.4.)

5.6.4.4 Errors

25(A) When search permission is denied for a component of the path prefix of *path,* then a call to *chmod(path, mode)* returns a value of (*int*)-1, sets *errno* to [EACCES], and does not change the file *mode.*

26({NAME_MAX}≤{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with [_POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of *path* is longer than the maximum number of bytes in a filename {NAME_MAX}, then a call to *chmod(path, mode)* returns a value of (*int*)-1 and sets *errno* to [ENAMETOOLONG].

27({NAME_MAX}>{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of *path* equals {PCTS_NAME_MAX}, then a call to chmod(path, mode) succeeds.

28(A) If {PATH_MAX} > {PCTS_PATH_MAX}:

When the length of the *path* argument exceeds the maximum number of bytes in a pathname {PATH_MAX}, then a call to *chmod(path, mode)* returns a value of (*int*)-1 and sets *errno* to [ENAMETOOLONG].

Otherwise:

When the length of the path argument is {PCTS_PATH_MAX}, then a call to *chmod*() is successful.

29(A) When a component of the *path* prefix of path is a not a directory, then a call to *chmod(path, mode)* returns a value of (*int*)-1 and sets *errno* to [ENOTDIR].

30(A) When the file named by *path* does not exist, then a call to *chmod(path, mode)* returns a value of (*int*)-1 and sets *errno* to [ENOENT].

31(A) When a component of the path prefix of *path* does not exist, then a call to *chmod(path, mode)* returns a value of (*int*)-1 and sets *errno* to [ENOENT].

32(A) When the argument *path* points to an empty string, then a call to *chmod(path, mode)* returns a value of (*int*)-1 and sets *errno* to [ENOENT].

33(A) When the effective user ID does not match the owner of the file and the calling process does not have the appropriate privileges to change the file mode, then a call to *chmod(path, mode)* returns a value of (*int*)-1, sets *errno* to [EPERM], and does not change the file *mode.*

34(C) If the implementation supports a read-only file system:

When the named file resides on a read-only file system, then a call to *chmod(path, mode)* returns a value of (*int*)-1, sets *errno* to [EROFS], does not change the mode of the file, and does not mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the file. (See GA13 in 2.3.5.)

5.6.5 *chown*()

5.6.5.1 Synopsis

01(A) If the implementation provides C Standard {2} support:

When the header `<unistd.h>` is included, then the function prototype `int chown(const char *, uid_t, gid_t)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<unistd.h>` is included. then the function *chown*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *chown*() is defined as a macro when the header `<unistd.h>` is included:

When the macro *chown*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *chown*() is defined as a macro in the header `<unistd.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

5.6.5.2 Description

04(A) When the first filename component of the *path* argument is ".", and the pathname does not begin with a slash, then *chown*() resolves the pathname by locating the second filename component (when specified) in the current working directory. (See GA14 in 2.3.6.)

05(PCTS_APP_OWNER && PCTS_ROOT_WRITABLE?A:UNTESTED)

When the *path* argument points to the string "`/`", then *chown*() resolves the pathname to the root directory of the process. (See GA15 in 2.3.6.)

06(PCTS_APP_OWNER && PCTS_ROOT_WRITABLE?A:UNTESTED)

When the *path* argument points to the string "`///`", then *chown*() resolves the pathname to the root directory of the process. (See GA16 in 2.3.6.)

07(A) When the *path* argument points to a string beginning with a single slash or beginning with three or more slashes, then *chown*() resolves the pathname by locating the first filename component of the pathname in the root directory of the process. (See GA17 in 2.3.6.)

08(A) When the first filename component of the *path* argument is "..", the pathname does not begin with a slash, and the current working directory is not the root directory of the process, then *chown*() resolves the pathname by locating the second filename component (when specified) in the parent directory of the current working directory. (See GA18 in 2.3.6.)

09(A) When the *path* argument points to the string "F1/" and `F1` is a directory, then *chown*() resolves the pathname to `F1`, which is in the current working directory. (See GA19 in 2.3.6.)

10(A) When the argument *path* points to the string "F1//" and F1 is a directory, then *chown*() resolves the pathname to `F1`, which is in the current working directory. (See GA20 in 2.3.6.)

11(A) When the *path* argument points to the string "F1/F2", then *chown*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA21 in 2.3.6.)

12(A) When the *path* argument points to the string "F1/./F2", then *chown*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA22 in 2.3.6.)

13(A) When the *path* argument points to the string "F1/../F1/F2", then *chown*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA23 in 2.3.6.)

14(A) When the *path* argument points to the string "F1//F2", then *chown*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA24 in 2.3.6.)

15(C) If {_POSIX_NO_TRUNC} is not supported in the corresponding directory:

When the *path* component is a string of more than {NAME_MAX} bytes in a directory for which {_POSIX_NO_TRUNC} is not supported, then *chown*() resolves the pathname component by truncating it to {NAME_MAX} bytes. (See GA25 in 2.3.6.)

16(A) On a call to *chown(path, owner, group)*, the pathname *path* supports *filenames* containing any of the following characters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ -

(See GA02 in 2.2.2.32.)

17(A) On a call to *chown(path, owner, group)*, the pathname *path* retains the unique identity of its upper- and lowercase letters. (See GA03 in 2.2.2.32.)

18(A) When search access is granted to each component of the path prefix of path, then the standard file access control mechanism does not cause *chown*() to indicate that file access is denied.

D01(C) If the implementation provides a mechanism to assign to a process the appropriate privileges to change file owner:

The details are confined in 2.2.2.4 or 5.6.5.2 of the PCD.1.

19(C) If the implementation provides a method for associating with a process the appropriate privilege to change the owner of a file:

When the process has the appropriate privileges to change the owner of a file, then a call to *chown(path, owner, group)* sets the user ID and group ID of the named file to the numeric values confined in *owner* and *group* respectively, and a value of zero is returned.

20(C) If the behavior associated with {_POSIX_CHOWN_RESTRICTED} is supported for the file:

When the effective user ID is equal to the user ID of the file, when the process does not have the appropriate privileges to change the owner of a file, when the argument owner is equal to the user ID of the file, and when the argument group is equal either to the effective group ID of the calling process or to one of its supplementary group IDs, then a call to *chown(path, owner, group)* sets the group ID of the file named by path to the numeric value contained in *group*, and a value of zero is returned.

21(C) If the behavior associated with {_POSIX_CHOWN RESTRICTED} is not supported for the file:

When the effective user ID matches the user ID of the current file, then a call to *chown(path, owner, group)* sets the user ID and group ID of the file named by *path* to the numeric values contained in *owner* and *group* respectively, and a value of zero is returned.

D02(C) If the implementation provides a mechanism to assign to a process the appropriate privileges to change file owner:

The effect on the S_ISUID and S_ISGID bits of the file mode of a regular file upon successful return from *chown*() from a process with appropriate privileges are contained in 2.2.2.4 or 5.6.5.2 of the PCD.1.

22(PCTS_CHMOD_SET_IDS?A:UNTESTED)

When the path argument refers to a regular file, and when the call is made by a process without the appropriate privileges to change the owner of a file, then a successful call to *chown(path, owner, group)* clears the set-user-ID and set-group-ID bits of the file mode.

23(A) A successful call to *chown*() marks for update the *st_crime* time-related field of the file.

5.6.5.3 Returns

R01 When a call to *chown*() completes successfully, then a value of (*int*)0 is returned. (See Assertions 19–21 in 5.6.5.2.)

R02 When a call to *chown*() completes unsuccessfully, then a value of (*int*)-1 is returned and sets *errno* to indicate the error, and no change is made to the owner and group of the file. (See Assertions 24, 25, and 27–37 in 5.6.5.4.)

5.6.5.4 Errors

24(A) When search permission is denied for a component of the path prefix of *path*, then a call to *chown*(*path*, *owner*, *group*) returns a value of (*int*)-1, sets *errno* to [EACCES], and does not change the owner and group of the file.

25({NAME_MAX}≤{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of *path* is longer than the maximum number of bytes in a filename {NAME_MAX}, then a call to *chown*(*path*, *owner*, *group*) returns a value of (*int*)-1 and sets *errno* to [ENAMETOOLONG].

26({NAME_MAX}>{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of *path* equals {PCTS_NAME_MAX}, then a call to *chown*(*path*, *owner*, *group*) succeeds.

27(A) If {PATH_MAX} ≤ {PCTS_PATH_MAX}:

When the length of the *path* argument exceeds the maximum number of bytes in a pathname {PATH_MAX}, then a call to *chown*(*path*, *owner*, *group*) returns a value of (*int*)-1 and sets *errno* to [ENAMETOOLONG].

Otherwise:

When the length of the path argument is {PCTS_PATH_MAX}, then a call to *chown*() is successful.

28(A) When a component of the path prefix of *path* is not a directory, then a call to *chown*(*path*, *owner*, *group*) returns a value of (*int*)-1 and sets *errno* to [ENOTDIR].

29(A) When the file named by *path* does not exist, then a call to *chown*(*path*, *owner*, *group*) returns a value of (*int*)-1 and sets *errno* to [ENOENT].

30(A) When a component of the path prefix of *path* does not exist, then a call to *chown*(*path*, *owner*, *group*) returns a value of (*int*)-1 and sets *errno* to [ENOENT].

31(A) When the *path* argument points to an empty string, then a call to *chown*(*path*, *owner*, *group*) returns a value of (*int*)-1 and sets *errno* to [ENOENT].

32(C) If the behavior associated with {_POSIX_CHOWN_RESTRICTED} is supported for the file

When the calling process does not have the appropriate privileges to change the owner of a file and the owner does not match the user ID of the file, then a call to *chown*(*path*, *owner*, *group*) returns a value of (*int*)-1, sets *errno* to [EPERM], and does not change the owner and group of the file.

33(C) If the behavior associated with {_POSIX_CHOWN_RESTRICTED} is supported for the file:

When the calling process does not have the appropriate privileges to change the owner of a file, and when the group does not match either the effective group ID of the process or any supplementary group ID, then a call to *chown*(*path*, *owner*, *group*) returns a value of (*int*)-1, sets *errno* to [EPERM], and does not change the *owner* and *group* of the file.

34(A) When the argument owner does not match the effective user ID, then a call to *chown*(*path*, *owner*, *group*) returns a value of (*int*)-1, sets *errno* to [EPERM], and does not change the owner and group of the file.

35(C) If the implementation supports a read-only file system:

When the named file resides on a read-only file system, then a call to *chown*(*path*, *owner*, *group*) returns a value of (*int*)-1, sets *errno* to [EROFS], does not change the owner and group of the file, and does not mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the file. (See GA13 in 2.3.5.)

D03(C) If the implementation supports the detection of [EINVAL] for *chown*():

The details under which [EINVAL] occurs for *chown*() are contained in 5.6.5.4 of the PCD.1. (See DGA02 in 2.4.)

36(PCTS_INVALID_OWNER?C:UNTESTED)

If the implementation provides the mechanism for creating a process with the appropriate privilege to change the owner and group ID of a file and the implementation supports the detection of [EINVAL] for *chown*():

When the calling process has the appropriate privileges to change the owner and group ID, and when either the owner or group ID supplied is invalid, then a call to *chown*(*path*, *owner*, *group*) returns a value of (*int*)-1, sets *errno* to [EINVAL], and does not change the owner and group of the file.

37(D) If the implementation provides the mechanism for creating a process with the appropriate privilege to change the owner and group ID of a file and the implementation does not support the detection of [EINVAL] for *chown*():

When the calling process has the appropriate privileges to change the owner or group ID, and when either the owner or group ID supplied is invalid, then a call to *chown*(*path*, *owner*, *group*) is successful (unless a different error condition is detected). (See GA26 in 2.4.)

See Reason 2 in Section 5. of POSIX.3 {4}.

5.6.6 *utime*()

5.6.6.1 Synopsis

01(A) If the implementation provides C Standard {2} support:

When the header `<utime.h>` is included, then the function prototype `int utime(const char *, const struct utimbuf *)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<utime.h>` is included, then the function *utime*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *utime*() is defined as a macro when the header `<utime.h>` is included:

When the macro *utime*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type int. (See GA37 in 2.7.3.)

03(C) If *utime*() is defined as a macro in the header `<utime.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

5.6.6.2 Description

04(A) When the first filename component of the *path* argument is ".", and the pathname does not begin with a slash, then *utime*() resolves the pathname by locating the second filename component (when specified) in the current working directory. (See GA14 in 2.3.6.)

05(PCTS_APP_TIMES && PCTS_ROOT_WRITABLE?A:UNTESTED)

When the *path* argument points to the string "`/`", then *utime*() resolves the pathname to the root directory of the process. (See GA15 in 2.3.6.)

06(PCTS_APP_TIMES && PCTS_ROOT_WRITABLE?A:UNTESTED)

When the *path* argument points to the string "`///`", then *utime*() resolves the pathname to the root directory of the process. (See GA16 in 2.3.6.)

07(A) When the *path* argument points to a string beginning with a single slash or beginning with three or more slashes, then *utime*() resolves the pathname by locating the first filename component of the pathname in the root directory of the process. (See GA17 in 2.3.6.)

08(A) When the first filename component of the *path* argument is "..", the pathname does not begin with a slash, and the current working directory is not the root directory of the process, then *utime*() resolves the pathname by locating the second filename component (when specified) in the parent directory of the current working directory. (See GA18 in 2.3.6.)

09(A) When the *path* argument points to the string "F1/" and `F1` is a directory, then *utime*() resolves the pathname to `F1`, which is in the current working directory. (See GA19 in 2.3.6.)

10(A) When the argument *path* points to the string "F1//" and F1 is a directory, then *utime*() resolves the pathname to `F1`, which is in the current working directory. (See GA20 in 2.3.6.)

11(A) When the *path* argument points to the string "F1/F2", then *utime*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA21 in 2.3.6.)

12(A) When the *path* argument points to the string "F1/./F2", then *utime*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA22 in 2.3.6.)

13(A) When the *path* argument points to the string "F1/../F1/F2", then *utime*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA23 in 2.3.6.)

14(A) When the *path* argument points to the string "F1/F2" then *utime*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA24 in 2.3.6.)

15(C) If {_POSIX_NO_TRUNC} is not supported in the corresponding directory:

When the *path* component is a string of more than {NAME_MAX} bytes in a directory for which {_POSIX_NO_TRUNC} is not supported, then *utime*() resolves the pathname component by truncating it to {NAME_MAX} bytes. (See GA25 in 2.3.6.)

16(A) On a call to *utime*(*path*, *times*), the pathname *path* supports *filenames* containing any of the following characters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ -

(See GA02 in 2.2.2.32.)

17(A) On a call to *utime*(*path*, *times*), the pathname *path* retains the unique identity of its upper- and lowercase letters. (See GA03 in 2.2.2.32.)

18(A) When *utime*(*path*, *times*) is granted write access to *path* and search access to the path prefix of *path*, then the standard file access control mechanism does not cause *utime*() to indicate that file access is denied.

19(A) When the *times* argument is NULL and the effective user ID matches the owner of the file, then a call to *utime*(*path*, *times*) sets the last accessed time and last modified time to the current time and returns a value of zero.

20(A) When the times argument is NULL and the process has write permission on the file, then a call to *utime*(*path*, *times*) sets the last accessed time and last modified time to the current time and returns a value of zero.

21(A) When the *times* argument is not NULL and addresses a valid *utimbuf* structure, and when the effective user ID matches the owner of the file, then a call to *utime*(*path*, *times*) sets the last accessed time and last modified time to the *actime* and *modtime* fields, respectively, of the structure *utimbuf* and returns a value of zero.

Testing Requirements:

When the *utimbuf* structure contains additional members beyond those specified in POSIX.1 {3}, and when the effect of these members is not activated, then these additional structure members do not affect the behavior of the call to *utime*().

22(C) If the implementation provides the mechanism for creating a process with the appropriate privilege to change the file times:

When the process has the appropriate privileges to change the file times, and when the effective user ID does not match the owner of the file, then a call to *utime*(*path*, *times*) sets the last accessed time and last modified time to the *actime* and *modtime* fields, respectively, of the structure *utimbuf* addressed by the non-NULL argument *times* and returns a value of zero

Testing Requirements:

When the *utimbuf* structure contains additional members beyond those specified in POSIX.1 {3}, and when the effect of these members is not activated, then these additional structure members do not affect the behavior of the call to *utime*().

23(C) If the implementation provides the mechanism for creating a process with the appropriate privilege to change the file times:

When the process has the appropriate privileges to change the file times, then a call to *utime*(*path*, *NULL*) sets the last accessed time and last modified time to the current time and returns the value zero.

24(A) When the header `<utime.h>` is included, then the elements *asctime* and *modtime* are defined in struct *utimbuf* as type *time_t*.

D01(C) If the *utimbuf* structure includes members other than those specified by POSIX.1 {3}, and this is documented:

The details are contained in 5.6.6.2 of the PCD.1.

25(A) A call to *utime*() marks for update the *st_ctime* time-related field of the file.

5.6.6.3 Returns

R01 When a call to *utime*() completes successfully, then a value of (*int*)0 is returned. (See Assertions 19–23 in 5.6.6.2.)

R02 When a call to *utime*() completes unsuccessfully, then a value of (*int*)−1 is returned, sets *errno* to indicate the error, and the file times are not affected. (See Assertions 26–28 and 30–36 in 5.6.6.4.)

5.6.6.4 Errors

26(A) When search permission is denied for a component of the path prefix of *path*, then a call to *utime*(*path*, *times*) returns a value of (*int*)-1, sets *errno* to [EACCES], and does not affect the file times.

27(A) When the times argument is NULL, when the effective user ID of the process does not match the owner of the file, and when write access is denied, then a call to *utime*(*path*, *times*) returns a value of (*int*)-1, sets *errno* to [EACCES], and does not affect the file times.

28({NAME_MAX}_≤{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of path is longer than the maximum number of bytes in a filename {NAME_MAX}, then a call to *utime*(*path*, *times*) returns a value of (*int*)-1 and sets *errno* to [ENAMETOOLONG].

29({NAME_MAX}>{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of *path* equals {PCTS_NAME_MAX}, then a call to *utime*(*path*, *times*) succeeds.

30(A) If {PATH_MAX} ≤ {PCTS_PATH_MAX}:

When the length of the *path* argument exceeds the maximum number of bytes in a pathname {PATH_MAX}, then a call to *utime*(*path*, *times*) returns a value of (*int*)-1 and sets *errno* to [ENAMETOOLONG].

Otherwise:

When the length of the path argument is {PCTS_PATH_MAX}, then a call to *utime*() is successful.

31(A) When the file named by *path* does not exist, then a call to *utime*(*path*, *times*) returns a value of (*int*)-1 and sets *errno* to [ENOENT].

32(A) When a component of the path prefix of *path* does not exist, then a call to *utime*(*path*, *times*) returns a value of (*int*)-1 and sets *errno* to [ENOENT].

33(A) When the *path* argument points to an empty string, then a call to *utime*(*path*, *times*) returns a value of (*int*)-1 and sets *errno* to [ENOENT].

34(A) When a component of the path prefix of *path* is not a directory, then a call to *utime*(*path*, *times*) returns a value of (*int*)-1 and sets *errno* to [ENOTDIR].

35(A) When the *times* argument is not NULL, when the effective user ID of the calling process has write access to the file but does not match the owner of the file, and when the calling process does not have the appropriate privileges to change the file times, then a call to *utime*(*path*, *times*) returns a value of (*int*)-1, sets *errno* to [EPERM], and does not affect the file times.

36(C) If the implementation supports a read-only file system:

When the named file resides on a read-only file system, then a call to *utime*(*path*, *times*) returns a value of (*int*)-1, sets *errno* to [EROFS], and does not mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the file. (See GA13 in 2.3.5.)

5.7 Configurable Pathname Variables

5.7.1 Get Configurable Pathname Variables

5.7.1.1 *pathconf*() (5.7.1)

5.7.1.1.1 Synopsis (5.7.1.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<unistd.h>` is included, then the function prototype `long pathconf(const char *, int)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<unistd.h>` is included, then the function *pathconf*() is declared with the result type *long*. (See GA36 in 2.7.3.)

02(C) If *pathconf*() is defined as a macro when the header `<unistd.h>` is included:

When the macro *pathconf*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *long*. (See GA37 in 2.7.3.)

03(C) If *pathconf*() is defined as a macro in the header `<unistd.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

5.7.1.1.2 Description (5.7.1.2)

04(B) When the first filename component of the *path* argument is ".", and the pathname does not begin with a slash, then *pathconf*() resolves the pathname by locating the second filename component (when specified) in the current working directory. (See GA14 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

05(B) When the path argument points to the string "`/`" then *pathconf*() resolves the pathname to the root directory of the process. (See GA15 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

06(B) When the *path* argument points to the string "`///`", then *pathconf*() resolves the pathname to the root directory of the process. (See GA16 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

07(B) When the *path* argument points to a string beginning with a single slash or beginning with three or more slashes, then *pathconf*() resolves the pathname by locating the first filename component of the pathname in the root directory of the process. (See GA17 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

08(B) When the first filename component of the *path* argument is "..", the pathname does not begin with a slash, and the current working directory is not the root directory of the process, then *pathconf*() resolves the pathname by locating the second filename component (when specified) in the parent directory of the current working directory. (See GA18 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

09(B) When the *path* argument points to the string "F1/" and `F1` is a directory, then *pathconf*() resolves the pathname to `F1`, which is in the current working directory. (See GA19 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

10(B) When the argument *path* points to the string "F1//" and `F1` is a directory, then *pathconf*() resolves the pathname to `F1`, which is in the current working directory. (See GA20 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

11(B) When the *path* argument points to the string "F1/F2", then *pathconf*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA21 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

12(B) When the *path* argument points to the string "F1//F2", then *pathconf*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA22 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

13(B) When the *path* argument points to the string "F1/../F1/F2", then *pathconf*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA23 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

14(B) When the *path* argument points to the string "F1//F2", then *pathconf*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA24 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

15(D) If {_POSIX_NO_TRUNC} is not supported in the corresponding directory:

When the *path* component is a string of more, than {NAME_MAX} bytes in a directory for which {_POSIX_NO_TRUNC} is not supported, then *pathconf*() resolves the pathname component by truncating it to {NAME_MAX} bytes. (See GA25 in 2.3.6.)

See Reason 3 in Section 5. of POSIX.3 {4}.

16(A) On a call to *pathconf*(*path*, *name*) the pathname *path* supports *filenames* containing any of the following characters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ -

(See GA02 in 5.1.1.)

17(B) On a call to *pathconf*(*path*, *name*), the pathnome *path* retains the unique identity of its upper- and lowercase letters. (See GA03 in 5.1.1.)

See Reason 3 in Section 5. of POSIX.3 {4}.

18(A) When *pathconf*(*path*, *name*) is granted search access to the path prefix of *path*, then the standard file access control mechanism does not cause *pathconf*() to indicate that file access is denied.

19(A) When the header `<unistd.>` is included, then the symbolic constants _PC_LINK_MAX, _PC_MAX_CANON, _PC_MAX_INPUT, _PC_NAME_MAX,_PC_PATH_MAX, _PC_PIPE_BUF, _PC_CHOWN_RESTRICTED, _PC_NO_TRUNC, and _PC_VDISABLE are defined and are distinct.

20(A) A call to *pathconf*(*path*, *_PC_LINK_MAX*) either returns a value greater than or equal to {_POSIX_LINK_MAX} or returns -1.

21(C) If LINK_MAX is defined when `<limits.h>` is included:

A call to *pathconf*(*path*, *_PC_LINK_MAX*) returns a value greater than or equal to the value defined for LINK_MAX in `<limits.h>`.

22(PCTS_GTI_DEVICE?A:UNTESTED)

When *path* refers to a terminal file, then a call to *pathconf*(*path*, *_PC_MAX_CANON*) either returns a value greater than or equal to {_POSIX_MAX_CANON} or returns −1.

23(PCTS_GTI_DEVICE?C:UNTESTED)

If MAX_CANON is defined when `<limits.h>` is included:

When *path* refers to a terminal file, then a call to *pathconf*(*path*, *_PC_MAX_CANON*) returns a value greater than or equal to the value defined for MAX_CANON in `<limits.h>`.

24(PCTS_GTI_DEVICE?A:UNTESTED)

When *path* refers to a terminal file, then a call to *pathconf*(*path*, *_PC_MAX_INPUT*) either returns a value greater than or equal to {_POSIX_MAX_INPUT} or returns -1.

25(PCTS_GTI_DEVICE?C:UNTESTED)

If MAX_INPUT is defined when `<limits.h>` is included:

When *path* refers to a terminal file, then a call to *pathconf*(*path*, *_PC_MAX_INPUT*) returns a value greater than or equal to the value defined for MAX_INPUT in `<limits.h>`.

26(A) When *path* refers to a directory, then a call to *pathconf*(*path*, *_PC_NAME_MAX*) either returns a value greater than or equal to {_POSIX_NAME_MAX} or returns −1.

27(C) If NAME_MAX is defined when <limits.h> is included:
When *path* refers to a directory, then a call to *pathconf*(*path, _PC_NAME_MAX*) returns a value greater than or equal to the value defined for NAME_MAX in <limits.h>.

28(A) When *path* refers to a directory, then the value returned by *pathconf*(*path, _PC_NAME_MAX*) applies to the filenames within the directory.

29(A) When *path* refers to a directory, then a call to *pathconf*(*path, PC_PATH_MAX*) either returns a value greater than or equal to {_POSIX_PATH_MAX} or returns -1.

30(C) If PATH MAX is defined when <limits.h> is included:
When *path* refers to a directory, then a call to *pathconf*(*path, PC_PATH_MAX*) returns a value greater than or equal to the value defined for PATH_MAX in <limits.h>.

31(A) When *path* refers to the current working directory, then the value returned by *pathconf*(*path, _PC_PATH_MAX*) is the maximum length of a relative pathname—the terminating null character need not be included in the length.

32(A) When *path* refers to a FIFO or directory, then a call to *pathconf*(*path, _PC_PIPE_BUF*) either returns a value greater than or equal to {_POSIX_PIPE_BUF} or returns -1.

33(C) If PIPE_BUF is defined when <limits.h> is included:
When *path* refers to a FIFO or directory, then a call to *pathconf*(*path, _PC_PIPE_BUF*) returns a value greater than or equal to the value defined for PIPE_BUF in <limits.h>.

34(A) When *path* refers to a FIFO, then the value returned by *pathconf*(*path, _PC_PIPE_BUF*) applies to the FIFO.

35(A) When *path* refers to a directory, then the value returned by *pathconf*(*path, _PC_PIPE_BUF*) applies to any FIFO that exists or can be created within the directory.

36(C) If _POSIX_CHOWN_RESTRICTED is defined when <unistd.h> is included:
A call to *pathconf*(*path, PC_CHOWN_RESTRICTED*) returns a value = {_POSIX_CHOWN_RESTRICTED}.

37(A) When *path* refers to a directory, then the value returned by *pathconf*(*path, PC_CHOWN_RESTRICTED*) applies to any files, other than directories, that exist or can be created within the directory.

38(C) If_POSIX_NO_TRUNC is defined when <unistd.h> is included:
When *path* refers to a directory, then a call to *pathconf*(*path, _PC_NO_TRUNC*) returns a value = {_POSIX_NO_TRUNC}.

39(A) When *path* refers to a directory, then the value returned by *pathconf*(*path, _PC_NO_TRUNC*) applies to the filenames within the directory.

40(PCTS_GTI_DEVICE?C:UNTESTED)

If_POSIX_VDISABLE is defined when <unistd.h> is included:
When *path* refers to a terminal file, then a call to *pathconf(path, _PC_VDISABLE*) returns the same value as that defined for _POSIX_VDISABLE in <unistd.h>.

41(PCTS_GTI_DEVICE?A:UNTESTED)

When *path* refers to a terminal, then the value returned by *pathconf(path, _PC_VDISABLE)* can be used to disable the effects associated with the receipt of a special control character from the terminal.

D01(C) If the implementation supports additional configurable pathname variables beyond those listed in Table 5.2 of POSIX.1 [3}, and this is documented:
The details are contained in 5.7.1.2 of the PCD.1.

D02(C) If the implementation supports the association of the variable names {MAX_CANON}, {MAX_INPUT}, or {_POSIX_VDISABLE} in a call to *pathconf*() with other than a terminal file, and this is documented:
The details are contained in 5.7.1.2 of the PCD.1.

D03(C) If the implementation supports the association of the variable names {NAME_MAX}, {PATH_MAX}, or {_POSIX_NO_TRUNC} in a call to *pathconf*() with file types other than a directory, and this is documented:
The details are contained in 5.7.1.2 of the PCD.1.

D04(C) If the implementation supports the association of the variable name {PIPE_BUF} in a call to *pathconf*() with file types other than a pipe, a FIFO, or a directory, and this is documented:
The details are contained in 5.7.1.2 of the PCD.1.

5.7.1.1.3 Returns (5.7.1.3)

R01 When a call to *pathconf*() completes successfully, then the current pathable value for the file or directory is returned without changing *errno*. (See Assertions 4–41 in 5.7.1.1.2.)

R02 When a call to *pathconf*() completes unsuccessfully, then a value of (*long*)–1 is returned and sets *errno* to indicate the error. (See Assertions 42–51 and 53–57 in 5.7.1.1.4.)

5.7.1.1.4 Errors (5.7.1.4)

D05(C) If the implementation supports the detection of [EINVAL] for *pathconf*():
The details under which [EINVAL] occurs for *pathconf*() are contained in 5.7.1.4 of the PCD.1. (See DGA02 in 2.4.)

42(A) When the value of *name* is invalid, then a call to *pathconf(path, name)* returns a value of (*long*)-1 and sets *errno* to [EINVAL].

D06(C) If the implementation supports the detection of [EACCES] for *pathconf*():
The details under which [EACCES] occurs for *pathconf*() are contained in 5.7.1.4 of the PCD.1. (See DGA02 in 2.4.)

43(A) When search permission is denied for a component of the path prefix of *path*, then a call to *pathconf*(*path, name*) either returns a value of ((*long*)-1) and sets *errno* to [EACCES] or returns the value associated with *name* for file named *path.*

44(A) When *path* does not refer to a terminal file, then a call to *pathconf(path, _PC_MAX_CANON)* either returns a value of ((*long*)-1) and sets *errno* to [EINVAL] or returns a valid value for {MAX_CANON}.

45(A) When *path* does not refer to a terminal file, then a call to *pathconf(path, _PC_MAX_INPUT)* either returns a value of ((*long*)-1) and sets *errno* to [EINVAL] or returns a valid value for {MAX_INPUT}.

46(A) When *path* does not refer to a directory, then a call to *pathconf(path, _PC_NAME_MAX)* either returns a value of ((*long*)-1) and sets *errno* to [EINVAL] or returns a valid value for {NAME_MAX}.

47(A) When *path* does not refer to a directory, then a call to *pathconf(path, _PC_PATH_MAX)* either returns a value of ((*long*)-1) and sets *errno* to [EINVAL] or returns a valid value for {PATH_MAX}.

48(A) When *path* does not refer to a FIFO or directory, then a call to *pathconf(path, _PC PIPEBUF)* either returns a value of ((*long*)-1) and sets *errno* to [EINVAL] or returns a valid value for {PIPE_BUF}.

49(A) When *path* does not refer to a directory, then a call to *pathconf(path, _PC_NO_TRUNC)* either returns a value of ((*long*)-1) and sets *errno* to [EINVAL] or returns a valid value for {_POSIX_NO_TRUNC}.

50(A) When *path* does not refer to a terminal file, then a call to *pathconf(path, _PC_VDISABLE)* either returns a value of ((*long*)-1) and sets *errno* to [EINVAL] or returns a valid value for {_POSIX_VDISABLE}.

D07(C) If the implementation supports the detection of [ENAMETOOLONG] for *pathconf*():
The details under which [ENAMETOOLONG] occurs for *pathconf*() are contained in 5.7.1.4 of the PCD.1. (See DGA02 in 2.4.)

51({NAME_MAX}≤{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX NO TRUNC} is supported for the file:

When the length of a pathname component of *path* is longer than the maximum number of bytes in a filename {NAME_MAX}, then a call to *pathconf(path, name)* either returns a value of ((*long*)-1) and sets *errno* to [ENAMETOOLONG] or returns a value appropriate for *name.*

52({NAME_MAX}>{PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the file:

When the length of a pathname component of *path* equals {PCTS_NAME_MAX}, then a call to *pathconf(path, name)* succeeds.

53(A) If {PATH_MAX} ≤ {PCTS_PATH_MAX}:

When the length of the *path* argument exceeds the maximum number of bytes in a pathname {PATH_MAX}, then a call to *pathconf(path, name)* either returns a value of ((*long*)-1) and sets *errno* to [ENAMETOOLONG] or returns a value appropriate for *name.*

Otherwise:

When the length of the path argument is {PCTS_PATH_MAX}, then a call to *pathconf*() is successful.

D08(C) If the implementation supports the detection of [ENOENT] for *pathconf*():

The details under which [ENOENT] occurs for *closedir*() are contained in 5.7.1.4 of the PCD.1. (See DGA02 in 2.4.)

54(A) When the file named by *path* does not exist, then a call to *pathconf(path, name)* either returns a value of ((*long*)-1) and sets *errno* to [ENOENT] or returns a value appropriate for *name.*

55(A) When a component of the path prefix of *path* does not exist, then a call to *pathconf(path, name)* either returns a value of ((*long*)-1) and sets *errno* to [ENOENT] or returns a value appropriate for *name.*

56(A) When the *path* argument points to an empty string. then a call to *pathconf(path, name)* either returns a value of ((*long*)-1) and sets *errno* to [ENOENT] or returns a value appropriate for *name*.

D09(C) If the implementation supports the detection of [ENOTDIR] for *pathconf*():

The details under which this error occurs are contained in 5.7.1.4 of the PCD.1. (See DGA02 in 2.4.)

57(A) When a component of the path prefix of *path* is not a directory, then a call to *pathconf(path, name)* either returns a value of ((*long*)-1) and sets *errno* to [ENOTDIR] or returns a value appropriate for *name.*

5.7.1.2 *fpathconf*() (5.7.1)

5.7.1.2.1 Synopsis (5.7.1.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<unistd.h>` is included, then the function prototype `long fpathconf(int, int)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<unistd.h>` is included, then the function *fpathconf*() is declared with the result type *long.* (See GA36 in 2.7.3.)

02(C) If *fpathconf*() is defined as a macro when the header `<unistd.h>` is included:

When the macro *fpathconf*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *long.* (See GA37 in 2.7.3.)

03(C) If *fpathconf*() is defined as a macro in the header `<unistd.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

5.7.1.2.2 Description (5.7.1.2)

04(A) A call to *fpathconf(fildes, _PC_LINK_MAX)* returns a value greater than or equal to {_POSIX_LINK_MAX}.

05(C) If LINK_MAX is defined when <limits.h> is included:

A call to *fpathconf(fildes, _PCLINK_MAX)* returns a value greater than or equal to the value defined for LINK_MAX in <limits.h>.

06(PCTS_GTI_DEVICE?A:UNTESTED)

When *fildes* refers to a terminal file, then a call to *fpathconf(fildes, _PC_MAX_CANON)* either returns a value greater than or equal to {_POSIX_MAX_CANON} or returns –1.

07(PCTS_GTI_DEVICE?C:UNTESTED)

If MAX_CANON is defined when <limits.h> is included:

When *fildes* refers to a terminal file, then a call to *fpathconf(fildes, PCMAX_CANON)* returns a value greater than or equal to the value defined for MAX_CANON in <limits.h>.

08(PCTS_GTI_DEVICE?A:UNTESTED)

When *fildes* refers to a terminal file, then a call to *fpathconf(fildes, _PC_MAX_INPUT)* either returns a value greater than or equal to {_POSIX_MAX_INPUT} or returns –1.

09(PCTS_GTI_DEVICE?C:UNTESTED)

If {MAX_INPUT} is defined when <limits.h> is included:

When *fildes* refers to a terminal file, then a call to *fpathconf(fildes, _PC_MAX_INPUT)* returns a value greater than or equal to the value defined for MAX_INPUT in <limits.h>.

10(A) When *fildes* refers to a directory, then a call to *fpathconfi(fildes, _PC_NAME_MAX)* either returns a value greater than or equal to {_POSIX_NAME_MAX} or returns –1.

11(C) If NAME_MAX is defined when <limits.h> is included:

When *fildes* refers to a directory, then a call to *fpathconf(fildes, _PC_NAME_MAX)* returns a value greater than or equal to the value defined for NAME_MAX in <limits.h>.

12(A) When *fildes* refers to a directory, then the value returned by *fpathconf(fildes, _PC_NAME_MAX)* applies to the filenames within the directory.

13(A) When *fildes* refers to a directory, then a call to *fpathconf(fildess, _PC_PATH_MAX)* either returns a value greater than or equal to {_POSIX_PATH_MAX} or returns –1.

14(C) If PATH_MAX is defined when <limits.h> is included:

When *fildes* refers to a directory, then a call to *fpathconf(fildes, _PC_PATH_MAX)* returns a value greater than or equal to the value defined for PATH_MAX in <limits.h>.

15(A) When *fildes* refers to the current working directory, then the value returned by *fpathconf(fildes, _PC_PATH_MAX)* is the maximum length of a relative pathname—the terminating null character need not be included in the length.

16(A) When *fildes* refers to a pipe, a FIFO, or a directory, then a call to *fpathconf(fildes, _PC_PIPE_BUF)* either returns a value greater than or equal to {_POSIX_PIPE_BUF} or returns -1.

Testing Requirements:

Test for file types of pipe, FIFO, and directory.

17(C) If PIPE_BUF is defined when <limits.h> is included:

When fildes refers to a pipe, a FIFO, or a directory, then a call to *fpathconf(fildes, _PC_PIPE_BUF)* returns a value greater than or equal to the value defined for PIPE_BUF in <limits.h>.

Testing Requirements:

Test for file types of pipe, FIFO, and directory.

18(A) When *fildes* refers to a pipe or a FIFO, then the value returned by *fpathconf(fildes, _PC_PIPE_BUF)* applies to the pipe or the FIFO.

Testing Requirements:

Test for file types of pipe and FIFO.

19(A) When *fildes* refers to a directory, then the value returned by *fpatheonf(fildes, _PC_PIPE_BUF)* applies to any FIFOs that exist or can be created within the directory.

20(C) If _POSIX_CHOWN_RESTRICTED is defined when `<unistd.h>` is included:

A call to *fpathconf(fildes, _PC_CHOWN_RESTRICTED*) returns a value equal to {_POSIX_CHOWN_RESTRICTED}.

21(A) When *fildes* refers to a directory, then the value returned by *fpathconf(fildes, _PC_CHOWN_RESTRICTED)* applies to all files other than directories that exist or can be created within the directory.

22(C) If {_POSIX_NO_TRUNC} is defined when `<unistd.h>` is included:

When *fildes* refers to a directory, then a call to *fpathconf(fildes, _PC_NO_TRUNC)* returns a value equal to {_POSIX_NO_TRUNC}.

23(A) When *fildes* refers to a directory, then the value returned by *fpathconfi(fildes, _PC_NO_TRUNC)* applies to the filenames within the directory.

24(PCTS_GTI_DEVICE?C:UNTESTED)

If {_POSIX_VDISABLE} is defined when `<unistd.h>` is included:

When fildes refers to a terminal file, then a call to *fpathconf(fildes, PC_VDISABLE)* returns the same value as that defined for _POSIX_VDISABLE in `<unistd.h>`.

25(PCTS_GTI_DEVICE?A:UNTESTED)

When *fildes* refers to a terminal, then the value returned by *fpathconf(fildes, _PC_VDISABLE)* can be used to disable the effects associated with the receipt of a special control character from the terminal.

D01(C) If the implementation supports the association of the variable names {MAX_CANON}, {MAX_INPUT}, or {_POSIX_VDISABLE} in a call to fpathconf() with a file type other than a terminal file, and this is documented:

The details are contained in 5.7.1.2 of the PCD.1.

D02(C) If the implementation supports the association of the variable names {NAME_MAX}, {PATH_MAX}, or {_POSIX_NO_TRUNC} in a call to fpathconf() with a file type other than a directory, and this is documented:

The details are contained in 5.7.1.2 of the PCD.1.

D03(C) If the implementation supports the association of the variable name {PIPE_BUF} in a call to *fpathconf*() with other than a pipe, a FIFO, or a directory, and this is documented:

The details are contained in 5.7.1.2 of the PCD.1.

5.7.1.2.3 Returns (5.7.1.3)

R01 When a call to *fpathconf*() completes successfully, then the current variable value for the file or directory is returned wit, hour changing *errno*. (See Assertions 4–25 in 5.7.1.2.2.)

R02 When a call to *fpathconf*() completes unsuccessfully, then a value of (*long*)–1 is returned and sets *errno* to indicate the error. (See Assertions 26–34 in 5.7.1.2.4.)

5.7.1.2.4 Errors (5.7.1.4)

26(A) When the value of *name* is invalid, then a call to *fpathconf(fildes, name)* returns a value of (*long*)-1 and sets *errno* to [EINVAL].

D04(C) If the implementation supports the detection of [EBADF] for *fpathconf*():
The details under which this error occurs are contained in 5.7.1.4 of the PCD.1. (See DGA02 in 2.4.)

27(A) When the *fildes* argument is not a valid file descriptor, then a call to *fpathconf(fildes, name)* either returns a value of ((*long*)-1) and sets *errno* to [EBADF] or returns a value associated with *name*.

D05(C) If the implementation supports the detection of [EINVAL] for *fpathconf*():
The details under which this error occurs are contained in 5.7.1.4 of the PCD.1. (See DGA02 in 2.4.)

28(A) When *fildes* does not refer to a terminal file, then a call to *fpathconf(fildes, _PC_MAX_CANON)* either returns a value of ((*long*)-1) and sets *errno* to [EINVAL] or returns a valid value for {MAX_CANON}.

29(A) When *fildes* does not refer to a terminal file, then a call to*fpathconf(fildes, _PC_MAX_INPUT)* either returns a value of ((*long*)-1) and sets *errno* to [EINVAL] or returns a valid value for {MAX_INPUT}.

30(A) When *fildes* does not refer to a directory, then a call to *fpathconf(fildes, _PC_NAME_MAX)* either returns a value of ((*long*)-1) and sets *errno* to [EINVAL] or returns a valid value for {NAME_MAX}.

31(A) When *fildes* does not refer to a directory, then a call to *fpathconf(fildes, _PC_PATH_MAX)* either returns a value of ((*long*)-1) and sets *errno* to [EINVAL] or returns a valid value for {PATH_MAX}.

32(A) When *fildes* does not refer to a pipe, a FIFO, or a directory, then a call to *fpathconf(fildes, _PC_PIPE_BUF)* either returns a value of ((*long*)−1) and sets *errno* to [EINVAL] or returns a valid value for {PIPE_BUF}.

33(A) When *fildes* does not refer to a directory, then a call to *fpathconf(fildes, _PC_NO_TRUNC*) either returns a value of ((*long*)-1) and sets *errno* to [EINVAL] or returns a valid value for {_POSIX_NO_TRUNC}.

34(A) When *fildes* does not refer to a terminal file, then a call to *fpathconf(fildes, _PC_VDISABLE*) either returns a value of ((*long*)-1) and sets *errno* to [EINVAL] or returns a valid value for {_POSIX_VDISABLE}.

6. Input and Output Primitives

6.1 Pipes

6.1.1 *pipe*()

6.1.1.1 Synopsis

01(A) If the implementation provides C Standard {2} support:
When the header `<unistd.h>` is included, then the function prototype `int pipe(int [2])` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<unistd.h>` is included, then the function *pipe*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *pipe*() is defined as a macro when the header `<unistd.h>` is included:
When the macro *pipe*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *pipe*() is defined as a macro in the header `<unistd.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

6.1.1.2 Description

04(A) A successful call to *pipe*(*fildes*) creates a pipe, places two file descriptors in the array *fildes*, and returns a value of zero.

05(A) When a call to *pipe*() completes successfully, then data can be written to *fildes[1]*.

06(A) When a call to *pipe*() completes successfully, then data can be read from *fildes[0]*.

07(A) The two file descriptors placed in the array *fildes* by a call to *pipe*(*fildes*) are the two lowest file descriptors available at the time of the *pipe*() call.

08(A) When a call to *pipe*() completes successfully, then the O_NONBLOCK flag is cleared on both file descriptors.

09(A) When a call to pipe completes successfully, then the FD_CLOEXEC flag is clear on both file descriptors.

10(A) A call to *read*() on *fildes[0]* accesses the data written to *fildes[1]* on a first-in-first-out basis.

11(A) A call to *pipe*() marks for update the *st_atime*, *st_ctime*, and *st_mtime* time-related fields of the pipe.

6.1.1.3 Returns

R01 When a call to *pipe*() completes successfully, then a value of (*int*)0 is returned. (See Assertion 4 in 6.1.1.2.)

R02 When a call to *pipe*() completes unsuccessfully, then a value of (*int*)-1 is returned and sets *errno* to indicate the error. (See Assertions 12 and 13 in 6.1.1.4.)

6.1.1.4 Errors

12(A) If {OPEN_MAX} ≤ {PCTS_OPEN_MAX}:

When {OPEN_MAX} -1 or more files are opened, then a call to *pipe*(*fildes*) returns a value of (*int*)-1 and sets *errno* to [EMFILE].

Otherwise:

{PCTS_OPEN_MAX} files can be opened.

13(B) When the number of open files in the system would exceed a system-imposed limit, then a call to *pipe*(*fildes*) returns a value of (*int*)-1 and sets *errno* to [ENFILE].

See Reason 1 in Section 5 of POSIX.3 {4}.

6.2 File Descriptor Manipulation

6.2.1 Duplicate an Open File Descriptor

6.2.1.1 *dup*() (6.2.1)

6.2.1.1.1 Synopsis (6.2.1.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<unistd.h>` is included, then the function prototype `int dup(int)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<unistd.h>` is included, then the function *dup*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *dup*() is defined as a macro when the header `<unistd.h>` is included:

When the macro *dup*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *dup*() is defined as a macro in the header `<unistd.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

6.2.1.1.2 Description (6.2.1.2)

04(A) A call to *dup*(*fildes*) returns a new file descriptor with the following attributes:
- It is the lowest numbered available file descriptor.
- It refers to the same open file description (file pointer, access mode, file status flags) as the original file descriptor.
- It shares the same locks as the original file descriptor.
- It has the FD_CLOEXEC flag clear.

6.2.1.1.3 Returns (6.2.1.3)

R01 When a call to *dup*() completes successfully, then the lowest numbered available file descriptor is returned. (See Assertion 4 in 6.2.1.1.2.)

R02 When a call to *dup*() completes unsuccessfully, then a value of (*int*)-1 is returned and sets *errno* to indicate the error (See Assertions 5 and 6 in 6.2.1.1.4.)

6.2.1.1.4 Errors (6.2.1.4)

05(A) When *fildes* is not a valid open file descriptor, then a call to *dup*(*fildes*) returns a value of (*int*)-1 and sets *errno* to [EBADF].

06(A) If {OPEN_MAX} ≤ {PCTS_OPEN_MAX}:
When {OPEN_MAX} files are open, then a call to *dup*(*fildes*) returns a value of (*int*)-1 and sets *errno* to [EMFILE].

Otherwise:
{PCTS_OPEN_MAX} files can be opened.

6.2.1.2 *dup2*() (6.2.1)

6.2.1.2.1 Synopsis (6.2.1.1)

01(A) If the implementation provides C Standard {2} support:
When the header `<unistd.h>` is included, then the function prototype `int dup2(int, int)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<unistd.h>` is included, then the function *dup2*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *dup2*() is defined as a macro when the header `<unistd.h>` is included:
When the macro *dup2*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *dup2*() is defined as a macro in the header `<unistd.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

6.2.1.2.2 Description (6.2.1.2)

04(A) When *fildes* is a valid file descriptor and is equal to *fildes2*, then a call to *dup2*(*fildes*, *fildes2*) returns *fildes2* without closing it.

05(A) A call to *dup2*(*fildes*, *fildes2*) returns a new file descriptor with the following attributes:

- Its value is equal to *fildes2*.
- It refers to the same open file description (file pointer, access mode, file status flags) as *fildes*.
- It shares the same locks as *fildes*.
- It has the FD_CLOEXEC flag clear.

6.2.1.2.3 Returns (6.2.1.3)

R01 When a call to *dup2*() completes successfully, then *fildes2* is returned. (See Assertion 4 in 6.2.1.2.2.)

R02 When a call to *dup2*(*fildes*, *fildes2*) completes unsuccessfully, then a value of (*int*)-1 is returned, sets *errno* to indicate the error, and the file referenced by *fildes2* is not closed. (See Assertion 6 in 6.2.1.2.4.)

6.2.1.2.4 Errors (6.2.1.4)

06(A) When *fildes* is not a valid open file descriptor, then a call to *dup2*(*fildes*, *fildes2*) returns a value of (*int*)-1, sets *errno* to [EBADF], and does not close argument *fildes2*.

07(A) When *fildes2* is negative, then a call to *dup2*(*fildes*, *fildes2*) returns a value of (*int*)-1 and sets *errno* to [EBADF].

08(A) If a call to *sysconf*(*_SC_OPEN_MAX*) does not return -1:

When *fildes2* is greater than or equal to {OPEN_MAX}, then a call to *dup2*(*fildes*, *fildes2*) returns a value of (*int*)-1 and sets *errno* to [EBADF].

Otherwise:

When *fildes2* takes the value {PCTS_OPEN_MAX}, then a call to *dup2*() does not give an [EBADF] error.

09(B) When *dup2*() is interrupted by a signal, then the call to *dup2*(*fildes*, *fildes2*) returns a value of (*int*)-1 and sets *errno* to [EINTR].

See Reason 3 in Section 5. of POSIX.3 {4}.

6.3 File Descriptor Deassignment

6.3.1 *close*()

6.3.1.1 Synopsis

01(A) If the implementation provides C Standard {2} support:

When the header `<unistd.h>` is included, then the function prototype `int close(int)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<unistd.h>` is included, then the function *close*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *close*() is defined as a macro when the header `<unistd.h>` is included:

When the macro *close*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *close*() is defined as a macro in the header `<unistd.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

6.3.1.2 Description

04(A) The *close*() function deallocates the file descriptor indicated by its argument (i.e., this file descriptor will be made available for subsequent *open*()s by the process), and a value of zero is returned.

05(A) When a close on the file descriptor is performed, then all outstanding record locks owned by the process on the file associated with the file descriptor are removed.

R01 When *close*() is interrupted by a signal that is to be caught, then the call to *close*() returns a value of (*int*)-1 and sets *errno* to [EINTR]. (See Assertion 13 in 6.3.1.4.)

D01(C) If the state of *fildes* when *close*() is interrupted by a signal that is caught is documented:
The details are contained in 6.3.1.2 of the PCD.1.

06(B) When all file descriptors associated with a pipe have been closed, then data remaining in the pipe is discarded.

See Reason 3 in Section 5. of POSIX.3 {4}.

07(A) When all file descriptors associated with a FIFO special file have been closed, then data remaining in the FIFO is discarded.

08(B) When all file descriptors associated with an open file description have been closed, then the open file description is freed.

See Reason 1 in Section 5. of POSIX.3 {4}.

09(B) When all file descriptors associated with a file have been closed and the link count of the file is zero, then the space occupied by the file is freed.

See Reason 1 in Section 5. of POSIX.3 {4}.

10(B) When all file descriptors associated with a file have been closed and the link count of the file is zero, then the file is no longer accessible.

See Reason 3 in Section 5. of POSIX.3 {4}.

11(A) When a file is closed by the last process that had it open, all time-related fields still marked for update are updated.

6.3.1.3 Returns

R02 When a call to *close*() completes successfully, then a value of (*int*)0 is returned. (See Assertion 4 in 6.3.1.2.)

R03 When a call to *close*() completes unsuccessfully, then a value of (*int*)-1 is returned and sets *errno* to indicate the error. (See Assertions 12 and 13 in 6.3.1.4.)

6.3.1.4 Errors

12(A) When the *fildes* argument is not a valid file descriptor, then a call to *close*(*fildes*) returns a value of (*int*)-1 and sets *errno* to [EBADF].

13(B) When *close*() is interrupted by a signal, then the call to *close*(*fildes*) returns a value of (*int*)-1 and sets *errno* to [EINTR].

See Reason 3 in Section 5. of POSIX.3 {4}.

6.4 Input and Output

6.4.1 *read*()

6.4.1.1 Synopsis

01(A) If the implementation provides C Standard {2} support:
When the header `<unistd.h>` is included, then the function prototype `ssize_t read(int, void *, size_t)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<unistd.h>` is included, then the function *read*() is declared with the result type *ssize_t*. (See GA36 in 2.7.3.)

02(C) If *read*() is defined as a macro when the header `<unistd.h>` is included:
When the macro *read*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *ssize_t*. (See GA37 in 2.7.3.)

03(C) If *read*() is defined as a macro in the header `<unistd.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

6.4.1.2 Description

04(A) A call to *read*(*fildes*, *buf*, *nbytes*) reads no more than *nbyte* bytes from the file associated with the open file descriptor *fildes* into the buffer pointed to by *buf* and returns the number of bytes read.

05(A) A call to *read*(*fildes*, *buf*, *0*) returns zero and does not move the file offset pointer.

06(A) On a regular file, a call to *read*(*fildes*, *buf*, *nbytes*) starts at a position in the file given by the file offset associated with *fildes*.

07(A) Before a call to *read*() returns, the file offset is incremented by the number of bytes actually read.

D01(C) If the value of the file offset, after a *read*() of a file that is not capable of seeking, is documented:
The details are contained in 6.4.1.2 of the PCD.1.

08(A) On a file not capable of seeking, a call to *read*() starts at the current position in the file.

09(A) When a *read*() on a regular file requests more bytes than exist, then the call to *read*() returns the number of bytes from the current position to the end-of-file.

10(A) When a *read*() on a pipe or FIFO requests more bytes than are available and at least one byte is available, then the call to *read*() returns the number of bytes that are available.

Testing Requirements:
Test for both a pipe and a FIFO.

R01 When *read*() is interrupted by a signal before any data is read, then the call to *read*() returns a value of (*int*)-1 and sets *errno* to [EINTR]. (See Assertion 31 in 6.4.1.4.)

D02(C) If the conditions under which *read*(), when interrupted by a signal after having successfully read some data, returns -1 and sets *errno* to [EINTR] or returns the number of bytes read are documented:
The details are contained in 6.4.1.2 of the PCD.1.

11(PCTS_GTI_DEVICE?C:UNTESTED)

If {PCD_READ_INTERRUPTED} is **TRUE**:
When *read*() to a terminal device file is interrupted by a signal after successfully reading some data, then the call to *read*() returns the number of bytes read.

12(PCTS_GTI_DEVICE?C:UNTESTED)

If {PCD_READ_INTERRUPTED} is **FALSE**:
When *read*() to a terminal device file is interrupted by a signal after successfully reading some data, then the call to *read*() returns a value of (*ssize_t*)-1 and sets *errno* to [EINTR].

13(PCTS_GTI_DEVICE?C:UNTESTED)

If {PCD_READ_INTERRUPTED} is not documented:
When *read*() to a terminal device file is interrupted by a signal after successfully reading some data, then the call to *read*() either returns the number of bytes read or returns a value of (*ssize_t*)-1 and sets *errno* to [EINTR].

14(D) If {PCD_READ_INTERRUPTED} is **TRUE** and the implementation supports character special files:
When *read*() to a terminal device file is interrupted by a signal after successfully reading some data, then the call to *read*() returns the number of bytes read.

See Reason 1 in Section 5. of POSIX.3 {4}.

15(D) If {PCD_READ_INTERRUPTED} is **FALSE** and the implementation supports character special files:
When *read*() to a terminal device file is interrupted by a signal after successfully reading some data, then the call to *read*() returns a value of (*ssize_t*)-1 and sets *errno* to [EINTR].

See Reason 1 in Section 5. of POSIX.3 {4}.

16(D) If {PCD_READ_INTERRUPTED} is not documented and implementation supports character special file:
When *read*() to a terminal device file is interrupted by a signal after successfully reading some data, then the call to *read*() either returns the number of bytes read or returns a value of (*ssize_t*)-1 and sets *errno* to [EINTR].

See Reason 1 in Section 5. of POSIX.3 {4}.

17(B) When *read*() is interrupted by a signal after successfully reading some data from a pipe or FIFO, then the call to *read*() returns the number of bytes read.

Testing Requirements:
Test for both a pipe and a FIFO.

See Reason 3 in Section 5. of POSIX.3 {4}.

18(A) When the file offset for a regular file is at or beyond the end-of-file, then a call to *read*() returns zero.

D03(A) When a call to *read*() has returned an end-of-file, the result of a subsequent *read*() request on device special files is contained 6.4.1.2 of the PCD.1.

D04(A) The result of a *read*() with a value of *nbyte* greater than {SSIZE_MAX} is contained in 6.4.1.2 of the PCD.1.

19(A) When no process has an empty pipe or FIFO open for writing, then a call to *read*() on the pipe or FIFO returns zero.

Testing Requirements:
Test for both a pipe and a FIFO.

R02 When some process has an empty pipe or FIFO open for writing and the O_NONBLOCK flag is set, then a call to *read*() on the pipe or FIFO returns a value of (*int*)–1 and sets *errno* to [EAGAIN]. (See Assertion 27 in 6.4.1.4.)

20(A) When some process has an empty pipe or FIFO open for writing and the O_NONBLOCK flag is clear, then a call to *read*() on the pipe or FIFO blocks until some data is written or until the pipe or FIFO is closed by all processes that had it opened for writing.

Testing Requirements:
Test for both a pipe and a FIFO.

R03 When the O_NONBLOCK flag is set on the file descriptor and the process would be delayed in the read operation, then a call to *read*() for this character special file returns a value of (*int*)-1 and sets *errno* to [EAGAIN]. (See Assertion 22 in 7.1.1.5.)

R04 When the O_NONBLOCK flag is set on the file descriptor and the process would be delayed in the read operation, then a call to *read*() for this special file returns a value of (*int*)–1 and sets *errno* to [EAGAIN]. (See Assertion 28 in 6.4.1.4.)

R05 When the O_NONBLOCK flag is clear on the file descriptor and no data is currently available, then a call to *read*() for this special file blocks until some data becomes available. (See Assertion 19 in 7.1.1.5.)

21(D) If the implementation supports block special files with nonblocking I/O:

When the O_NONBLOCK flag is clear on a block special file descriptor and no data is currently available, then a call to *read*() for this special file blocks until some data becomes available.

See Reason 1 in Section 5. of POSIX.3 {4}.

R06 When the O_NONBLOCK flag is set on the character special file descriptor and data is currently available, then a call to *read*() for this special file returns the available data up to the number of bytes requested. (See Assertions 20 and 21 in 7.1.1.5.)

22(D) If the implementation supports block special files with nonblocking I/O:

When the O_NONBLOCK flag is set on the block special file descriptor and data is currently available, then a call to *read*() for this special file returns the available data up to the number of bytes requested.

See Reason 1 in Section 5. of POSIX.3 {4}.

23(A) When the file pointer precedes the end-of-file and data has not been written to that part of the file, then a call to *read*() from a regular file returns bytes with value zero.

Testing Requirements:

Test using *lseek*() and *fseek*().

24(A) A successful call to *read*() of more than zero bytes marks for update the *st_atime* field of the file.

25(A) A successful call to *read*() with nbytes set to zero does not mark for update the *st_atime* field of the file.

26(A) When *nbyte* is greater than zero, then a successful call to *read*(), which does not read any data from the file, marks for update the *st_atime* field of the file.

6.4.1.3 Returns

R07 When a call to *read*() completes successfully, then the number of bytes read is returned. (See Assertion 4 in 6.4.1.2.)

R08 When a call to *read*() completes unsuccessfully, then a value of (*ssize_t*)-1 is returned and sets *errno* to indicate the error. (See Assertions 27–33 in 6.4.1.4.)

6.4.1.4 Errors

27(A) When the file type is a pipe or FIFO, when the O_NONBLOCK flag is set for the file descriptor, and when the process would be delayed in the read operation, then a call to *read*() returns a value of (*ssize_t*)-1 and sets *errno* to [EAGAIN].

Testing Requirements:

Test for both a pipe and a FIFO.

R09 When the O_NONBLOCK flag is set on a character special file for a file that supports nonblocking I/O, and when the proceeds would be delayed in the read operation, then a call to *read*() for this special file returns a value of (*ssize_t*)-1 and sets *errno* to [EAGAIN]. (See Assertion 21 in 7.1.1.5.)

28(D) If the implementation supports block special files with nonblocking I/O:

When the O_NONBLOCK flag is set on such a block special file descriptor, and when the process would be delayed in the read operation, then a call to *read*() for this special file returns a value of (*ssize_t*)-1 and sets *errno* to [EAGAIN].

See Reason 1 in Section 5. of POSIX.3 {4}.

29(A) When the file descriptor argument *fildes* is not a valid file descriptor, then a call to *read*(*fildes*, *buf*, *nbytes*) returns a value of (*ssize_t*)-1 and sets *errno* to [EBADF].

30(A) When the file descriptor argument *fildes* is not open for reading, then a call to *read*(*fildes*, *buf*, *nbytes*) returns a value of (*ssize_t*)-1 and sets *errno* to [EBADF].

31(A) When *read*() is terminated due to receipt of a signal before any data is read, then the call to *read*() returns a value of (*ssize_t*)-1 and sets *errno* to [EINTR].

D05(C) If the implementation documents whether there are conditions other than those specified by POSIX.1 {3} under which *read*() sets *errno* to [EIO]:

The details are contained in 6.4.1.4 of the PCD.1.

32(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When a process in a background process group is attempting to *read*() from its controlling terminal, and when the process is ignoring or blocking the SIGTTIN signal, then the call to *read*() returns a value of (*ssize_t*)-1 and sets *errno* to [EIO].

33(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When a process in a background process group is attempting to *read*() from its controlling terminal, and when the process group of the process is orphaned, then the call to *read*() returns a value of (*ssize_t*)-1 and sets *errno* to [EIO].

6.4.2 *write*()

6.4.2.1 Synopsis

01(A) If the implementation provides C Standard {2} support:

When the header `<unistd.h>` is included, then the function prototype `ssize_t write(int, const void *, size t)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<unistd.h>` is included, then the function *write*() is declared with the result type *ssize_t*. (See GA36 in 2.7.3.)

02(C) If *write*() is defined as a macro when the header `<unisrd.h>` is included:

When the macro *write*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *ssize_t*. (See GA37 in 2.7.3.)

03(C) If *write*() is defined as a macro in the header `<unistd.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

6.4.2.2 Description

04(A) A call to *write*(*fildes*, *buf*, *nbyte*) writes *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the open file descriptor argument *fidles* and returns the number of bytes written.

05(A) When the file is a regular file, then a call to *write*(*times*, *buf*, *O*) returns zero, does not mark for update the *st_ctime* and *st_mtime* fields of the file, and no bytes are written.

Testing Requirements:

Test for both O_APPEND flag clear and O_APPEND flag set. The latter case will ensure that an attempt to append zero bytes to the file will not result in a change of the file offset.

D01(C) If the results of a call to *write*() when *nbyte* is zero and the file is not a regular file are documented:
The details are contained in 6.4.2.2 of the PCD.1.

06(A) When the file is a regular file, or another file capable of seeking, then a call to *write*(*fildes, buf, name*) starts writing at a position in the file given by the file offset associated with the file descriptor argument *fildes.*

07(A) Before a successful return from a call to *write*(), the file offset is incremented by the number of bytes actually written.

08(A) When the file is a regular file, and when the current file offset after a successful return from a call to *write*() is greater than the length of the file before the call, then the length of the file after the call is set to this file offset.

09(A) When the file is not capable of seeking, then a call to *write*() starts at the current position in the file.

D02(C) If the value of the file offset after a call to *write*() for a file type that is not capable of seeking is documented:
The details are contained in 6.4.2.2 of the PCD.1.

10(A) When the O_APPEND flag of the file status flags is set, then a call to *write*() sets the file offset to the end of the file prior to each *write*().

Testing Requirements:
Test for files opened in O_WRONLY and O_RDWR modes.

11(B) When the file is a regular file, and when a *write*() requests that more bytes be written than there is room for, then the call to *write*() writes only as many bytes for which there is room.

See Reason 4 in Section 5. of POSIX.3 {4}.

R01 When *write*() is interrupted by a signal before any data is written, then the call to *write*() returns a value of (*ssize_t*)-1 and sets *errno* to [EINTR]. (See Assertion 39 in 6.4.2.4.)

D03(C) If the conditions under which *write*(), when interrupted by a signal after it has successfully written some data, returns -1 and sets *errno* to [EINTR] or returns the number of bytes read are documented:
The details are contained in 6.4.2.2 of the PCD.1.

12(PCTS_GTI_DEVICE?C:UNTESTED)

If {PCD_WRITE_INTERRUPTED} is **TRUE**:
When *write*() to a terminal device file is interrupted by a signal after successfully writing some data, then the call to *write*() returns the number of bytes written.

13(PCTS_GTI_DEVICE?C:UNTESTED)

If {PCD_WRITE_INTERRUPTED} is **FALSE**:
When *write*() to a terminal device file is interrupted by a signal after successfully writing some data, then the call to *write*() returns a value of (*ssize_t*)-1 and sets *errno* to [EINTR].

14(PCTS_GTI_DEVICE?C:UNTESTED)

If {PCD_WRITE_INTERRUPTED} is not documented:
When *write*() to a terminal device file is interrupted by a signal after successfully writing some data, then the call to *write*() either returns the number of bytes written or returns a value of (*ssize_t*)-1 and sets *errno* to [EINTR].

15(D) If {PCD_WRITE_INTERRUPTED} is **TRUE** and the implementation supports character special files:
When *write*() to a character special file is interrupted by a signal after successfully writing some data, then the call to *write*() returns the number of bytes written.

See Reason 1 in Section 5. of POSIX.3 {4}.

16(D) If {PCD WRITE_INTERRUPTED} is **FALSE** and the implementation supports character special files:

When *write*() to a character special file is interrupted by a signal after successfully writing some data, then the call to *write*() returns a value of (*ssize_t*)-1 and sets *errno* to [EINTR].

See Reason 1 in Section 5. of POSIX.3 {4}.

17(D) If {PCD_WRITE_INTERRUPTED} is not documented and the implementation supports character special files:

When *write*() to a character special file is interrupted by a signal after successfully writing some data, then the call to *write*() either returns the number of bytes written or returns a value of (*ssize_t*)-1 and sets *errno* to [EINTR].

See Reason 1 in Section 5. of POSIX.3 {4}.

D04(A) When the value of *nbyte* is greater than {SSIZE_MAX}, the details describing the results of a call to *write*() are contained in 6.4.2.2 of the PCD.1.

18(A) When a *write*() to a regular file has returned successfully, then a successful *read*() from any byte position that was modified by the previous *write*() returns the data written to that position by that previous *write*() until such byte positions are again modified.

19(A) When a regular file already contains data at the position referenced by a successful call to *write*(), then the data at the position referenced are overwritten.

20(B) When the file is a pipe or a FIFO, when *write*()has transferred some data, and when *nbyte* is less than or equal to {PIPE_BUF}, then the call to *write*(*fildes, buf, nbyte*) does not return with *errno* set to [EINTR].

Testing Requirements:

Test for both a pipe and a FIFO.

See Reason 2 in Section 5. of POSIX.3 {4}.

21(A) When the file is a pipe or a FIFO, then a call to *write*() appends to the end of the pipe or FIFO.

Testing Requirements:

Test for both a pipe and a FIFO.

22(B) When the file is a pipe or a FIFO and *nbyte* is less than or equal to {PIPE_BUF}, then the call to *write*() does not interleave with data from other processes doing writes on the same pipe or FIFO.

Testing Requirements:

Test for both a pipe and a FIFO.

See Reason 3 in Section 5. of POSIX.3 {4}.

R02 When the file is a pipe or a FIFO, when the O_NONBLOCK flag is set, and when no data can be accepted at the time of the *write*(), then the call to *write*() does not block the process. (See Assertion 28 in 6.4.2.2.)

23(D) If the implementation supports character special files with nonblocking I/O:

When the file is a character special file, when the O_NONBLOCK flag is set, and when no data can be accepted at the time of the *write*(), then the call to *write*() does not block the process.

NOTE — The case of a terminal device file is covered by Assertion 43 in 7.1.1.8.

See Reason 1 in Section 5. of POSIX.3 {4}.

24(D) If the implementation supports block special files with nonblocking I/O:

When the file is a block special file, when the O_NONBLOCK flag is set, and when no data can be accepted at the time of the *write*(), then the call to *write*() does not block the process.

See Reason 1 in Section 5. of POSIX.3 {4}.

25(A) When the file is a pipe or a FIFO, when the O_NONBLOCK flag is set, when *nbyte* is less than or equal to {PIPE_BUF}, and when space exists in the pipe or FIFO for *nbyte* bytes of data, then a call to *write*(*fildes, buf, nbyte*) succeeds completely and returns *nbyte*.

Testing Requirements:

Test for both a pipe and a FIFO. If {PIPE_BUF} is greater than {PCTS_PIPE_BUF}, test with values of *nbyte* up to and including {PCTS_PIPE_BUF}.

26(A) When the file is a pipe or a FIFO, when the O_NONBLOCK flag is clear, and when *nbyte* is less than or equal to {PIPE_BUF}, then a call to *write*(*fildes*, *buf*, *nbyte*) blocks until space is available to complete the *write*() or the *write*() is interrupted by a signal.

Testing Requirements:

If {PIPE_BUF} is greater than {PCTS_PIPE_BUF}, test with values of *nbyte* up to and including {PCTS_PIPE_BUF}.

27({PIPE_BUF} ≤ {PCTS_PIPE_BUF}?A:UNTESTED)

When the file is a pipe or a FIFO, when the O_NONBLOCK flag is set, when *nbyte* is greater than {PIPE_BUF} bytes, and when at least one byte can be written, then a call to *write*() transfers what it can and returns the number of bytes written.

Testing Requirements:

Test for both a pipe and a FIFO.

28({PIPE_BUF} ≤ {PCTS_PIPE_BUF}?A:UNTESTED)

When the file is a pipe or a FIFO, when the O_NONBLOCK flag is set, when *nbyte* is greater than {PIPE_BUF} bytes, and when no data can be written, then a call to *write*() returns a value of (*ssize_t*)-1, sets *errno* to [EAGAIN], and transfers no data.

Testing Requirements:

Test for both a pipe and a FIFO.

29({PIPE_BUF} ≤ {PCTS_PIPE_BUF}?A:UNTESTED)

When the file is a pipe or a FIFO, when the O_NONBLOCK flag is set, when *nbyte* greater than {PIPE_BUF} bytes, and when all data previously written to the pipe or FIFO has been read, then a call to *write*(*fildes*, *buf*, *nbyte*) transfers at least {PIPE_BUF} bytes.

Testing Requirements:

Test for both a pipe and a FIFO.

30(D) If the implementation supports character special files with nonblocking I/O:

When the file is a character special file, when the O_NONBLOCK flag is clear, and when the file cannot accept the data immediately, then a call to *write*() blocks until the data can be accepted.

NOTE — The case of a terminal device file is covered by Assertion 43 in 7.1.1.8.

See Reason 1 in Section 5. of POSIX.3 {4}.

31(D) If the implementation supports block special files with nonblocking I/O:

When the file is a block special file, when the O_NONBLOCK flag is clear, and when the file cannot accept the data immediately, then a call to *write*() blocks until the data can be accepted.

See Reason 1 in Section 5. of POSIX.3 {4}.

32(D) If the implementation supports character special files with nonblocking I/O:

When the file is a character special file, when the O_NONBLOCK flag is set, and when some data can be written without blocking the process, then a call to *write*() either writes what it can and returns the number of bytes written, or returns a value of (*ssize_t*)-1, sets *errno* to [EAGAIN], and transfers no data.

NOTE — The case of a terminal device file is covered by Assertion 44 in 7.1.1.8.

See Reason 1 in Section 5. of POSIX.3 {4}.

33(D) If the implementation supports block special files with nonblocking I/O:

When the file is a block special file, when the O_NONBLOCK flag is set, and when some data can be written without blocking the process, then a call to *write*() either writes what it can and returns the number of bytes written, or returns a value of (*ssizet*)-1, sets *errno* to [EAGAIN], and transfers no data.

See Reason 1 in Section 5. of POSIX.3 {4}.

34(A) A successful call to *write*() of more than zero bytes marks for update the *st_ctime* and *st_mtime* fields of the file.

6.4.2.3 Returns

R03 When a call to *write*() completes successfully, then the number of bytes written is returned. (See Assertion 4 in 6.4.2.2.)

R04 When a call to *write*() completes unsuccessfully, then a value of (*ssize_t*)-1 is returned and sets *errno* to indicate the error. (See Assertions 35–42 in 6.4.2.4.)

6.4.2.4 Errors

35(A) When the file is a pipe or a FIFO, when the O_NONBLOCK flag is set, when *nbyte* is less than or equal to {PIPE_BUF}, and when insufficient capacity exists to accept the data, then a call to *write*() returns a value of (*ssize_t*)-1 and sets *errno* to [EAGAIN].

Testing Requirements:

Test for both a pipe and a FIFO. If {PIPE_BUF} is greater than {PCTS_PIPE_BUF}, test with values of *nbyte* up to and including {PCTS_PIPE_BUF}.

36(A) When the file descriptor argument *fildes* is not a valid file descriptor, then a call to *write*(*fildes, buf, nbyte*) returns a value of (*ssize_t*)-1 and sets *errno* to [EBADF].

37(A) When the file descriptor argument *fildes* is not open for writing, then a call to *write*(*fildes, buf, nbyte*) returns a value of (*ssize_t*)-1 and sets *errno* to [EBADF].

38(D) If the implementation supports a maximum file size:

When an attempt is made to write a file that would exceed an implementation-defined maximum size, then a call to *write*() returns a value of (*ssize_t*)-1 and sets *errno* to [EFBIG].

See Reason 3 in Section 5. of POSIX.3 {4}.

39(A) When *write*() is terminated due to receipt of a signal and no data was transferred, then the call to *write*() returns a value of (*ssize_t*)-1 and sets *errno* to [EINTR].

R05 When *write*() is interrupted by a signal after successfully writing some data, then the call to *write*() either returns the number of bytes written or returns a value of (*ssize_t*)-1 and sets *errno* to [EINTR]. (See Assertions 12–17 in 6.4.2.2.)

D05(C) If the implementation documents whether there are conditions other than those specified by POSIX.1 {3} under which *write*() sets *errno* to [EIO]:

The details are contained in 6.4.2.4 of the PCD.1.

40(PCTS_GTI_DEVICE:C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When the process is a member of an orphaned background process group attempting to *write*() to its controlling terminal, when TOS-TOP is set, and when the process is neither ignoring nor blocking the SIGTTOU signal, then a call to *write*() returns a value of (*ssize_t*)-1 and sets *errno* to [EIO].

41(B) When a device has no more space for data, then a call to *write*() on this device returns a value of (*ssize_t*)-1 and sets *errno* to [ENOSPC].

See Reason 4 in Section 5. of POSIX.3 {4}.

42(A) When the file is a pipe or a FIFO and the file is not open for reading by any process, then a call to *write*() returns a value of (*ssize_t*)-1, sets *errno* to [EPIPE], and sends a SIGPIPE signal to the calling process.

Testing Requirements:
Test for both a pipe and a FIFO.

6.5 Control Operations on Files

6.5.1 Data Definitions for File Control Operations

01(A) When the header `<fcntl.h>` is included, then the symbolic constants O_CREAT, O_EXCL, O_TRUNC, O_NOCTTY, O_APPEND, O_NONBLOCK, O_RDONLY, O_RDWR, O_WRONLY, and O_ACCMODE are defined.

02(A) When the header `<fcntl.h>` is included, then the symbolic constants F_DUPFD, F_GETFD, F_GETLK, F_SETFD, F_GETFL, F_SETFL, F_SETLK, and F_SETLKW are defined, and their values are unique, numbers.

03(A) When the header `<fcntl.h>` is included, then the symbolic constant FD_CLOEXEC is defined.

04(A) When the header `<fcntl.h>` is included, then the symbolic constants F_RDLCK, F_UNLCK, and F_WRLCK are defined, and their values are unique numbers.

05(A) When the header `<fcntl.h>` is included, then the set of symbolic constants O_RDONLY, O_WRONLY and O_RDWR are defined, and their values are unique.

06(A) When the header `<fcntl.h>` is included, then the set of symbolic constants O_ACCMODE, O_APPEND, O_CREAT, O_EXCL, O_NONBLOCK, O_NOCTTY, and O_TRUNC are defined, and their values are bitwise discrete.

07(A) When the header `<fcntl.h>` is included, then O_ACCMODE is defined as a mask for O_RDONLY, O_RDWR, and O_WRONLY.

6.5.2 *fcntl* ()

6.5.2.1 Synopsis

01(A) If the implementation provides C Standard {2} support:
When the header `<fcntl.h>` is included, then the function prototype
`int fcntl(int, int, ...)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<fcntl.h>` is included, then the function *fcntl*() is either declared with result type int or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *fcntl*() is defined as a macro when the header `<fcntl.h>` is included:
When the macro *fcntl*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *fcntl*() is defined as a macro in the header `<fcntl.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

6.5.2.2 Description

04(A) A call to *fcntl*(*fildes, F_DUPFD, arg*) provides the following attributes for the new file descriptor created:

— It is the lowest numbered available file descriptor greater than or equal to argument *arg*.
— It refers to the same open file description (file pointer, access mode, file status flags) as the original file descriptor.
— It shares the same locks as the original file descriptor.
— It has the FD_CLOEXEC flag clear.

05(A) A call to *fcntl*(*fildes, F_GETFD*) returns the status of the file descriptor flags.

06(A) When the flag in *arg* corresponding to FD_CLOEXEC is nonzero, then a call to *fcntl*(*fildes, F SETFD, arg*) sets the close-on-exec flag, and the file associated with *fildes* is closed after a successful `exec`.

07(A) When the flag in *arg* corresponding to FD_CLOEXEC is zero, then a call to *fcntl*(*fildes, F SETFD, arg*) clears the close-on-exec flag, and the file associated with *fildes* stays open across `exec` type functions.

08(A) A call to *fcntl*(*fildes, F_GETFL*) returns the file status flags O_APPEND and O_NONBLOCK, and the file access modes O_RDONLY, O_RDWR, and O_WRONLY.

09(A) A call to *fcntl*(*fildes, F_SETFL, O*) sets the status flags to 0.

10(A) A call to *fcntl*(*fildes, F. SETFL, O_NONBLOCK*) sets the status flags to O_NONBLOCK.

11(A) A call to *fcntl*(*fildes, F_SETFL, O_APPEND*) sets the status flags to O_APPEND.

12(A) A call to *fcntl*(*fildes, F SETFL, O_APPEND* | *O_NONBLOCK*) sets the status flags for O_APPEND and O_NONBLOCK.

13(A) When the third argument *arg* only contains the file status flags O_APPEND, O_NONBLOCK; the file access modes O_RDONLY, O_RDWR, O_WRONLY; and the *oflag* values O_CREAT, O_EXCL, O_NOCTTY, and O_TRUNC; then a call to *fcntl*(*fildes, F_SETFL, arg*) ignores any bits corresponding to the file access modes and the *oflag* values that are set in *arg*.

D01(C) If the implementation supports additional file status flags for the F_SETFL command for *fcntl*(), and this is documented:

The details on these additional file status flags are contained in 6.5.2.2 of the PCD.1.

D02(C) If advisory record locking is supported or unsupported for files other than regular files, and this is documented:

The details are contained in 6.5.2.2 of the PCD.1.

14(A) When a lock is present that blocks the request and *arg* is a pointer to the type *struct flock*, then a call to *fcntl*(*fildes, F_GETLK, arg*) returns the first such lock that blocks the lock description pointed to by arg. All elements of the *struct flock* addressed by *arg* are set, and the lock type is placed in the *struct flock* addressed by *arg* with the value of *l_whence* set to SEEK_SET.

Testing Requirements:

Test for all three values of *l_whence*.

15(A) When no lock exists in the region specified in the *struct flock* addressed by *arg* that would prevent the lock from being created, then a call to *fcntl*(*fildes, F_GETLK, arg*) sets the lock type in the structure pointed to by *arg* to F_UNLCK and does not change the other *struct flock* elements.

16(A) When the lock type in the structure pointed to by *arg* is set to F_RDLCK, then a call to *fcntl*(*fildes, F_SETLK, arg*) establishes a shared lock for the region specified in the *struct flock* addressed by *arg*.

17(A) When the lock type in the structure pointed to by *arg* is set to F_WRLCK, then a call to *fcntl*(*fildes, F_SETLK, arg*) establishes an exclusive lock for the region specified in the *struct flock* addressed by *arg*.

18(A) When the lock type in the structure pointed to by *arg* is set to F_UNLCK, then a call to *fcntl*(*fildes, F SETLK, arg*) removes any locks for the region specified in the *struct flock* addressed by *arg.*

19(A) When a shared or exclusive lock cannot be set, then a call to *fcntl*(*fildes, F_SETLK, arg*) returns without waiting.

20(A) When lock type in the structure pointed to by *arg* is set to F_RDLCK, then a call to *fcntl*(*fildes, F_SETLKW, arg*) establishes a shared lock for the region specified in the *struct flock* addressed by *arg*.

21(A) When the lock type in the structure pointed to by *arg* is set to F_WRLCK, then a call to *fcntl*(*fildes, F_SETLKW, arg*) establishes an exclusive lock for the region specified in the *struct flock* addressed by *arg*.

22(A) When the lock type in the structure pointed to by *arg* is set to F_UNLCK, then a call to *fcntl*(*fildes, F_SETLKW, arg*) removes any locks for the region specified in the *struct flock* addressed by *arg*.

23(A) When the shared or exclusive lock is blocked by other locks, then a call to *fcntl*(*fildes, F_SETLKW, arg*) waits until the request can be satisfied.

R01 When *fcntl*() is interrupted while waiting to set a lock on a region, then a call to *fcntl*(*fildes, F_SETLKW, arg*) returns a value of (*int*)-1 and sets *errno* to [EINTR]. (See Assertion 40 in 6.5.2.4.)

24(A) When the header `<fcntl.h>` is included, then the *struct flock* is defined and has the element names and types specified in Table 6.1.

Table 6.1—*flock* Structure

Element	Type
l_type	short
l_whence	short
l-start	off_t
l_len	off_t
l_pid	pid_t

25(A) When a shared lock has been set on a segment of a file, then other processes are able to set shared locks on that segment or a portion of it.

26(A) A shared lock prevents any other process from setting an exclusive lock on any portion of the part of the file to which the shared lock is applied.

R02 A request for a shared lock will fail if the file descriptor was not opened with read access. (See Assertion 38 in 6.5.2.4.)

27(A) An exclusive lock prevents any other process from setting a shared lock or an exclusive lock on any portion of the protected area.

R03 A request for an exclusive lock fails if the file descriptor was not opened with write access. (See Assertion 39 in 6.5.2.4.)

D03(C) If the action taken on calls to *fcntl*() for file locking when *l_len* is negative is documented:
The details are contained in 6.5.2.2 of the PCD.1.

R04 The *l_pid* field is only used with F_GETLK to return the process ID of the process holding a blocking lock. (See Assertion 14 in 6.5.2.2.)

28(A) Locks can start and extend beyond the current end-of-file.

29(A) When *l_len* is set to zero, then all bytes from the position specified by *l_whence* and *l_start* up to the end-of-file irrespective of the current file length are locked or unlocked as specified by the structure pointed to by *arg*.

30(A) Before a successful return from a F_SETLK or F_SETLKW request, the previous lock type for each byte in the specified region is replaced by the new lock type. The calling process has only one type of lock set for each byte in the file.

31(A) All locks associated with a file for a given process areremoved when a file descriptor for that file is closed.

32(A) When a process terminates, then all locks associated with a file for the process are removed.

33(A) Locks are not inherited by a child process created using the *fork*() function.

6.5.2.3 Returns

R05 When a call to *fcntl*(*fildes, F_DUPFD, arg*) completes successfully, then the lowest file descriptor greater than or equal to arg is returned. (See Assertion 4 in 6.5.2.2.)

R06 When a call to *fcntl*(*fildes, F_GETFD*) completes successfully, then the value of the close_on_exec flag is returned. (See Assertion 5 in 6.5.2.2.)

R07 When *cmd* are the command values F_SETFD, F_GETFL, F_SETFL, F_GETLK, F_SETLK, and F_SETLKW, and when the call to *fcntl*(*fildes, cmd*) completes successfully, then a value other than -1 is returned. (See Assertions 6–23 in 6.5.2.2.)

R08 When a call to *fcntl*() completes unsuccessfully, then a value of (*int*)−1 is returned and sets *errno* to indicate the error. (See Assertions 34–52 in 6.5.2.4.)

6.5.2.4 Errors

34(A) When the locking structure pointed to by *arg* contains F_RDLCK, and when some portion of the segment of a file to be locked is already exclusive-locked by another process, then a call to *fcntl*(*fildes, F_SETLK, arg*) returns a value of (*int*)-1 and sets *errno* to [EACCES] or [EAGAIN].

35(A) When the locking structure pointed to by *arg* contains F_WRLCK, and when some portion of the segment of a file to be locked is already exclusive-locked by another process, then a call to *fcntl*(*fildes, F_SETLK, arg*) returns a value of (*int*)-1 and sets *errno* to [EACCES] or [EAGAIN].

36(A) When the locking structure pointed to by arg contains F_WRLCK, and when some portion of the segment of the file to be locked is already shared-locked by another process, then a call to *fcntl*(*fildes, F_SETLK, arg*) returns a value of (*int*)-1 and sets *errno* to [EACCES] or [EAGAIN].

37(A) When the file descriptor argument is not a valid file descriptor, then a call to *fcntl*(*fildes, cmd, arg*) returns a value of (*int*)-1 and sets *errno* to [EBADF].

38(A) When the type of lock is a F_RDLCK lock but *fildes* is not a valid file descriptor open for reading, then a call to *fcntl*(*fildes, F_SETLK, arg*) and *fcntl*(*fildes, F_SETLKW, arg*) returns a value of (*int*)-1 and sets *errno* to [EBADF].

39(A) When the type of lock is a F_WRLCK lock but *fildes* is not a valid file descriptor open for writing, then a call to *fcntl*(*fildes, F_SETLK, arg*) and *fcntl*(*fildes, F_SETLKW, arg*) returns a value of (*int*)-1 and sets *errno* to [EBADF].

40(A) When *fcntl*() is interrupted by a signal, then a call to *fcntl*(*fildes, F_SETLKW, arg*) returns a value of (*int*)-1 and sets *errno* to [EINTR].

41(A) When *arg* is negative, then a call to *fcntl*(*fildes, F_DUPFD, arg*) returns a value of (*int*)−1 and sets *errno* to [EINVAL].

42(A) If a call to *sysconf*(*_SC_OPEN_MAX*) does not return -1:

When *arg* is greater than or equal to {OPEN_MAX}, then a call to *fcntl*(*fildes, F_DUPFD, arg*) returns a value of (*int*)-1 and sets *errno* to [EINVAL].

Otherwise:

When *arg* takes the value {PCTS_OPEN_MAX}, then a call to *fcntl*(*fildes, F_DUPFD, arg*) does not give an [EINVAL] error.

43(A) When the data to which *arg* points references a position prior to the beginning of the file, then a call to *fcntl*(*fildes, F_GETLK, arg*), *fcntl*(*fildes, F_SETLK, arg*), or *fcntl*(*fildes, F_SETLKW, arg*) returns a value of (*int*)−1 and sets *errno* to [EINVAL].

44(A) When the *l_type* field of the structure pointed to by *arg* is invalid, then a call to *fcntl*(*fildes, F_GETLK, arg*), *fcntl*(*fildes, F_SETLK, arg*), or *fcntl*(*fildes, F_SETLKW, arg*) returns a value of (*int*)-1 and sets *errno* to [EINVAL].

45(A) When the *l_whence* field of the structure pointed to by *arg* is invalid, then a call of *fcntl*(*fildes, F_GETLK, arg*), *fcntl*(*fildes, F_SETLK, arg*), or *fcntl*(*fildes, F_SETLKW, arg*) returns a value of (*int*)-1 and sets *errno* to [EINVAL].

46(C) If {PCD_NO_LOCK_FILE TYPE} is **TRUE**:

When the *fildes* refers to a file that does not support locking, then a call of *fcntl*(*fildes, F_GETLK, arg*), *fcntl*(*fildes, F_SETLK, arg*), or *fcntl*(*fildes, F_SETLKW, arg*) returns a value of (*int*)-1 and sets *errno* to [EINVAL].

47(C) If {PCD_NO_LOCK_FILE_TYPE} is not documented:

When the lock type in the structure pointed to by arg is set to F_RDLCK, then a call to *fcntl*(*fildes, F_SETLK, arg*) either establishes a shared lock for the region specified in the *struct flock* addressed by arg or returns a value of (*int*)-1 and sets *errno* to [EINVAL].

Testing Requirements:

Test for file types of FIFO and character special files.

48(A) If {OPEN_MAX} ≤ {PCTS_OPEN_MAX}:

When {OPEN_MAX} files have been opened, then a subsequent call to *fcntl*(*fildes, F_DUPFD, arg*) returns a value of (*int*)-1 and sets *errno* to [EMFILE].

Otherwise:

{PCTS_OPEN_MAX} files can be opened.

49(A) When no file descriptor greater than or equal to *arg* is available, then a call of *fcntl*(*fildes, F_DUPFD, arg*) returns a value of (*int*)-1 and sets *errno* to [EMFILE].

50(B) When satisfying the lock or unlock request that would result in the number of locked regions exceeding a system-imposed limit, then a call to *fcntl*(*fildes, F_SETLK, arg*) or *fcntl*(*fildes, F_SETLKW, arg*) returns a value of (*int*)-1 and sets *errno* to [ENOLCK].

See Reason 3 in Section 5. of POSIX.3 {4}.

D04(C) If the implementation supports the detection of [EDEADLK] for *fcntl*():

The details under which [EDEADLK] occurs for *fcntl*() are contained in 6.5.2.4 of the PCD.1. (See DGA02 in 2.4.)

51(C) If the implementation supports the detection of [EDEAELK] for *fcntl*():

When a deadlock condition occurs, then a call to *fcntl*(*fildes, F_SETLKW, arg*) returns a value of (*int*)-1 and sets *errno* to [EDEADLK].

52(D) If the implementation does not support the detection of [EDEADLK] for *fcntl*():

A call to *fcntl*(*fildes, F_SETLKW, arg*) is successful (unless adifferent error condition is detected). (See GA26 in 2.4.)

See Reason 2 in Section 5. of POSIX.3 {4}.

6.5.3 *lseek* ()

6.5.3.1 Synopsis

01(A) If the implementation provides C Standard {2} support:

When the header `<unistd.h>` is included, then the function prototype `off_t lseek(int, off_t, int)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<unistd.h>` is included, then the function *lseek*() is declared with the result type *off_t*. (See GA36 in 2.7.3.)

02(C) If *lseek*() is defined as a macro when the header `<unistd.h>` is included:

When the macro *lseek*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *off_t*. (See GA37 in 2.7.3.)

03(C) If *lseek*() is defined as a macro in the header `<unistd.h>`:

It evaluates its arguments only once, fully protected by parentheseswhen necessary, and protects its result value with extraparentheses when necessary. (See GA01 in 1.3.4.)

6.5.3.2 Description

04(A) When the file is capable of seeking, then a call to *lseek*(*fildes*, *offset*,*SEEK_SET*) sets the file *offset* offset bytes from the beginning of the file.The resulting offset location as measured in bytes from the beginning ofthe file is returned.

05(A) When the file is capable of seeking, then a call to *lseek*(*fildes*, *offset*,*SEEK_CUR*) sets the file offset offset bytes from the current position.The resulting offset location as measured in bytes from the beginning ofthe file is returned.

06(A) When the file is capable of seeking, then a call to *lseek*(*fildes*, *offset*, *SEEK_END*) sets the file offset to the size of the file plus *offset* bytes. The resulting offset location as measured in bytes from the beginning of the file is returned.

07(A) When the header `<unistd.h>` is included, then the symbolic constants SEEK_SET, SEEK_CUR, and SEEK_END are defined.

D01(A) The details on the behavior of the *lseek*() function on devices incapable of seeking are contained in 6.5.3.2 of the PCD.1.

R01 When the file is capable of seeking, and when the file offset has been set beyond the end of existing data in the file by a call to *lseek*() followed by a call to *write*(), then data read from gaps in the file—areas in the file where no data has been written—appear as though bytes with the value zero had been written. (See Assertion 18 in 6.4.1.2.)

08(A) When the file is capable of seeking, then a call to *lseek*() does not, by itself, extend the size of the file.

6.5.3.3 Returns

R02 When a call to *lseek*() completes successfully, then the resulting offset location as measured in bytes from the beginning of the file is returned. (See Assertions 4-6 in 6.5.3.2.)

R03 When a call to *lseek*() completes unsuccessfully, then a value of (*off t*)-1 is returned, sets *errno* to indicate the error, and the file pointer remains unchanged. (See Assertions 9–13 in 6.5.3.4.)

6.5.3.4 Errors

09(A) When the file descriptor argument is not a valid file descriptor, then a call to *lseek*(*fildes*, *offset*, *whence*) returns a value of (*off_t*)-1 and sets *errno* to [EBADF].

10(A) When the file is capable of seeking and the *whence* argument is not valid, then a call to *lseek*(*fildes*, *offset*, *whence*) returns a value of (*off_t*)-1, sets *errno* to [EINVAL], and does not alter the value of the file pointer.

11(A) When the file is capable of seeking and the resulting file offset would be invalid, then a call to *lseek*(*fildes*, *offset*, *whence*) returns a value of (*off_t*)-1, sets *errno* to [EINVAL], and does not alter the value of the file pointer.

12(A) When the *fildes* argument is associated with a pipe, then a call to *lseek*(*fildes*, *offset*, *whence*) returns a value of (*off_t*)-1, sets *errno* to [ESPIPE], and does not alter the value of the file pointer.

13(A) When the *fildes* argument is associated with a FIFO then a call to *lseek*(*fildes*, *offset*, *whence*) returns a value of (*off_t*)-1, sets *errno* to [ESPIPE], and does not alter the value of the file pointer.

7. Device- and Class-Specific Functions

Testing Requirements:

The assertions in this section are required to be tested on any device that supports the general terminal interface. This includes devices that control asynchronous communications ports and may include devices that control network connections or synchronous ports. An implementation need not provide any device that supports the general terminal interface, in which case the tests for these assertions will not be executed.
Many of the assertions in this chapter reference particular flags associated with part of the *termios* structure. These references do not explicitly state that these flags need to be set in the *termios* control structure that is in effect for the terminal device file that the test is using to verify the assertion. Unless otherwise stated in the assertion, this shall be the meaning associated with any statement in the assertion regarding the setting of flags contained in the *termios* structure.

7.1 General Terminal Interface

D01(A) The device types supported by the general terminal interface and whether network connections and asynchronous ports or both are supported by the general terminal interface are contained in 7.1 of the PCD.1.

7.1.1 Interface Characteristics

7.1.1.1 Opening a Terminal Device File

There are no assertions specific to this subclause.

7.1.1.2 Process Groups

R01 When all the processes in the foreground process group have terminated, the system does not change an existing background process group to be the foreground process group, but allows the terminal to have no foreground process group. (See Assertion 5 in 7.2.3.3.)

D02(C) If it is documented whether the terminal has a foreground process group when there is no longer any process whose process group ID matches the process group ID of the foreground process group, but there is a process whose process ID matches:

The details are contained in 7.1.1.2 of the PCD.1.

7.1.1.3 The Controlling Terminal

01(PCTS_GTI_DEVICE?A:UNTESTED)

Each process of a session that has a controlling terminal has the same controlling teminal.

02(PCTS_GTI_DEVICE?B:UNTESTED)

A controlling terminal associated with the current session cannot become the controlling terminal for another session.

See Reason 1 in Section 5. of POSIX.3 {4}.

03(PCTS_GTI_DEVICE?A:UNTESTED)

When a process that is not a session leader *open*()s a terminal file, then the terminal does not become the controlling terminal of the calling process.

04(PCTS_GTI_DEVICE?A:UNTESTED)

When a process uses the O_NOCTTY option on a call to *open*(), then the terminal does not become the controlling terminal of the calling process.

D03(A) When a session leader without a controlling terminal opens a terminal device file that is not already associated with a session—without using the O_NOCTTY option—the details on whether the terminal becomes the controlling terminal of the session leader are contained in 7.1.1.3 of the PCD.1.

05(PCTS_GTI_DEVICE?A:UNTESTED)

When a controlling terminal becomes associated with a session, then its foreground process group is set to the process group of the session leader.

R02 The controlling terminal is inherited by child processes during a *fork*() function. (See Assertion 20 in 3.1.1.2.)

06(PCTS_GTI_DEVICE?A:UNTESTED)

When a process creates a new session with the *setsid*() function, then it relinquishes its controlling terminal.

D04(C) If it is documented whether, upon the close of the last file descriptor in the system associated with the controlling terminal, all processes that have that terminal as their controlling terminal cease to have a controlling terminal:

The details are contained in 7.1.1.3 of the PCD.1.

07(PCTS_GTI_DEVICE?A:UNTESTED)

When a process closes all file descriptors associated with its controlling terminal, and when at least one other process has an open file descriptor that references the same controlling terminal, then the first process does not relinquish its controlling terminal.

D05(C) If it is documented whether a session leader can reacquire a controlling terminal after it has relinquished its controlling terminal because all file descriptors associated with that terminal have been closed:

The details are contained in 7.1.1.3 of the PCD.1.

08(PCTS_GTI_DEVICE?A:UNTESTED)

When a controlling process terminates, then the controlling terminal is disassociated from the current session, allowing it to be acquired by a new session leader.

09(PCTS_GTI_DEVICE?C:UNTESTED)

If, after a controlling process terminates, access to the terminal by processes in the session of the terminated process is denied:

Attempts to access the terminal by surviving processes in that session are treated as if modem disconnect had occurred.

Testing Requirements:

Test when *read*() returns 0 and *write*() returns -1 and sets *errno* to [EIO].

D06(C) If it is documented whether, after a controlling process terminates, access to the terminal by processes in the session of the terminated process is denied:

The details are contained in 7.1.1.3 of the PCD.1.

D07(A) The details how the controlling terminal for a session is allocated by the session leader are contained in 7.1.1.3 of the PCD.1.

7.1.1.4 Terminal Access Control

R03 A process in a foreground process group is allowed to read from its controlling terminal. (See Assertions 19-28 in 7.1.1.5 through 7.1.1.7.)

10(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When a member of a background process group that is not orphaned and is not blocking or ignoring the SIGTTIN signal reads from its controlling terminal, then a SIGTTIN signal is sent to each process in its process group.

11(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When a member of a background process group that is ignoring the SIGTTOU signal attempts to read from its controlling terminal, then a call to *read*() returns a value of (*ssize_t*)-1, sets *errno* to [EIO], and does not send the SIGTTIN signal to any process in its process group.

12(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When a member of a background process group that is blocking the SIGTTIN signal attempts to read from its controlling terminal, then the call to *read*() returns a value of (*ssize_t*)-1, sets *errno* to [EIO], and does not send the SIGTTIN signal to any process in its process group.

13(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} :is supported:

When a member of an orphaned background process group attempts to read from its controlling terminal, then the call to *read*() returns a value of (*ssize_t*)-1, sets *errno* to [EIO], and does not send the SIGTTIN signal to any process in its process group.

14(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB CONTROL} is supported:

When TOSTOP is set, and when a member of a background process group that is not orphaned and is not blocking or ignoring the SIGTTOU signal writes to its controlling terminal, then a SIGTTOU signal is sent to each process in its process group.

15(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB CONTROL} is supported:

When TOSTOP is clear, and when a member of a background process group, whether orphaned or not, attempts to write to its controlling terminal, then the call to *write*() succeeds, and the SIGTTOU signal is not sent to any process in its process group.

16(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When TOSTOP is set, and when a member of a background process group that is ignoring the SIGTTOU signal attempts to write to its controlling terminal, then the call to *write*() succeeds.

17(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When TOSTOP is set, and when a member of a background process group that is blocking the SIGTTOU signal attempts to write to its controlling terminal, then the call to *write*() succeeds.

18(PCTS_GTI_DEVICE?C :UNTESTE D)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When TOSTOP is set, and when a process that is a member of an orphaned background process group and is neither ignoring nor blocking SIGTTOU attempts to write to its controlling terminal, then the call to *write*() returns a value of (*ssize_t*)-1, sets *errno* to [EIO], and does not send the SIGTTOU signal to any process in its process group.

7.1.1.5 Input Processing and Reading Data

D08(C) If the system imposes a limit, {MAX_INPUT}, on the number of bytes that may be stored in a terminal input queue:

The behavior of the system when that limit is exceeded is documented in 7.1.1.5 of the PCD.1.

19(PCTS_GTI_DEVICE?A:UNTESTED)

When O_NONBLOCK is clear, then a call to *read*() is blocked until data is available or a signal has been received.

20(PCTS_GTI_DEVICE?A:UNTESTED)

When O_NONBLOCK is set, and when there is enough data available to satisfy the entire request, then a call to *read*() completes successfully, without blocking, reading all the requested data and returning the number of bytes read.

Testing Requirements:

Test for both canonical and noncanonical mode input processing.

21(PCTS_GTI_DEVICE?A:UNTESTED)

When O_NONBLOCK is set, and when there is not enough data available to satisfy the entire request, then a call to *read*() completes successfully, without blocking, reading as much data as possible, and returns the number of bytes it was able to read.

Testing Requirements:

Test for both canonical and noncanonical mode input processing.

22(PCTS_GTI_DEVICE?A:UNTESTED)

When O_NONBLOCK is set and there is no data available, then a call to *read*() completes without blocking, returns a value of (*ssize_t*)-1, and sets *errno* to [EAGAIN].

Testing Requirements:

Test for both canonical and noncanonical mode input processing.

7.1.1.6 Canonical Mode Input Processing

23(PCTS_GTI_DEVICE?A:UNTESTED)

When ICANON is set and O_NONBLOCK is not set, then a call to *read*() from a terminal does not return until an entire line has been typed or a signal has been received. A line is delimited by NL, EOF, or EOL.

Testing Requirements:

Test for NL, EOF, and EOL.

24(PCTS_GTI_DEVICE?A:UNTESTED)

When ICANON is set, then a call to *read*() from a terminal returns at most one line, even if more bytes are requested.

25(PCTS_GTI_DEVICE?A:UNTESTED)

When ICANON is set, then a call to *read*() requesting fewer bytes than available in the input line receives the number of bytes requested. The first byte received from the next *read*() is the first unread byte in the line, and no input data is lost.

26(PCTS_GTI_DEVICE?A:UNTESTED)

When ICANON is set, then the canonical input queue of a terminal is able to contain a minimum number of bytes.

Testing Requirements:

When {MAX_CANON} is greater than {PCTS_MAX_CANON}, then the minimum number of bytes the canonical queue can contain is {PCTS_MAX_CANON}; otherwise, it can contain {MAX_CANON} bytes.

D09(C) If the system imposes a limit, {MAX_CANON}, on the number of bytes in a line for a particular terminal device:

The behavior of the system when the terminal is processing input in canonical mode and the limit is exceeded is documented in 7.1.1.6 of the PCD.1.

R04 Erase and kill processing occurs when either of the two special characters, ERASE and KILL, is received. (See Assertions 52 and 53 in 7.1.1.9.)

7.1.1.7 Noncanonical Mode Input Processing

27(PCTS_GTI_DEVICE?A:UNTESTED)

When ICANON is clear, then all bytes input from a terminal are entered into the input queue and are not assembled into lines.

28(PCTS_GTI_DEVICE?A:UNTESTED)

When ICANON is clear, then erase and kill processing do not occur. ERASE and KILL are entered into the input queue.

D10(C) If the response to *read*() when MIN is greater than {MAX_INPUT} is documented:

The details are contained in 7.1.1.7 of the PCD.1.

7.1.1.7.1 CASE A: MIN > 0, TIME > 0

29(PCTS_GTI_DEVICE?A:UNTESTED)

When ICANON is clear, MIN is greater than 0, and TIME is greater than 0, then TIME acts as an interbyte timer of 0.1 s granularity, activated after the first byte is received.

30(PCTS_GTI_DEVICE?A:UNTESTED)

When ICANON is clear, MIN is greater than 0, and TIME is greater than 0, then the interbyte timer is reset after each byte is received.

31(PCTS_GTI_DEVICE?A:UNTESTED)

When ICANON is clear, MIN is greater than 0, TIME is greater than 0, and when MIN bytes are received before the interbyte timer expires, then the *read*() is satisfied.

32(PCTS_GTI_DEVICE?A:UNTESTED)

When ICANON is clear, MIN is greater than 0, TIME is greater than 0, and when the interbyte timer expires before MIN bytes are received, then the bytes received to that point are returned from the *read*().

33(PCTS_GTI_DEVICE?A:UNTESTED)

When ICANON is clear, MIN is greater than 0, and TIME is greater than 0, then a call to *read*() blocks until the MIN and TIME mechanisms are activated by the first byte received or by a signal being received.

34(PCTS_GTI_DEVICE?A:UNTESTED)

When ICANON is clear, MIN is greater than 0, TIME is greater than 0, and data is in the buffer at the time of the call to *read*(), then the timer is started as if data had been received immediately after the call to *read*().

7.1.1.7.2 CASE B: MIN > 0, TIME = 0

35(PCTS_GTI_DEVICE?A:UNTESTED)

When ICANON is clear, MIN is greater than 0, and TIME is equal to 0, then a call to *read*() blocks until MIN bytes are received or until a signal is received.

7.1.1.7.3 CASE C: MIN = 0, TIME > 0

36(PCTS_GTI_DEVICE?A:UNTESTED)

When ICANON is clear, MIN is equal to 0, and TIME is greater than 0, then a call to *read*() is satisfied when any data is received before TIME × 0,1 s pass.

37(PCTS_GTI_DEVICE?A:UNTESTED)

When ICANON is clear, MIN is equal to 0, TIME is greater than 0, the timer expires after TIME × 0,1 s, and the read queue is empty, then zero bytes are returned from the call to *read*().

38(PCTS_GTI_DEVICE?A:UNTESTED)

When ICANON is clear, MIN is equal to 0, TIME is greater than 0, and data is in the buffer at the time of the call to *read*(), then the timer is started as if data had been received immediately after the call to *read*().

7.1.1.7.4 CASE D: MIN = 0, TIME = 0

39(PCTS_GTI_DEVICE?A:UNTESTED)

When ICANON is clear, MIN is equal to 0, TIME is equal to 0, and there is enough data available to satisfy the request, then a call to *read*() returns the bytes requested.

40(PCTS_GTI_DEVICE?A:UNTESTED)

When ICANON is clear, MIN is equal to 0, TIME is equal. to 0, and there is some data available but less that the amount requested, then a call to *read*() returns the number of bytes currently available.

41(PCTS_GTI_DEVICE?A:UNTESTED)

When ICANON is clear, MIN is equal to 0, TIME is equal to 0, and there is no data available, then a call to *read*() returns zero bytes.

7.1.1.8 Writing Data and Output Processing

D11(C) If support or nonsupport for the condition when the implementation buffers *write*() output to a terminal device is documented:

The details are contained in 7.1.1.8 of the PCD.1.

42(PCTS_GTI_DEVICE?A:UNTESTED)

When O_NONBLOCK is set by *open*() or *fcntl*(), and when no data can be accepted at the time of the *write*(), then a call to *write*() does not block the process, returns a value of (*ssize_t*)-1, sets errno to [EAGAIN], and transfers no data.

43(PCTS_GTI_DEVICE?A:UNTESTED)

When O_NONBLOCK is clear, then a call to *write*() is blocked until the data can be accepted or a signal has been received.

44(PCTS_GTI_DEVICE?B:UNTESTED)

When O_NONBLOCK is set and some data can be written, then a call to *write*() writes what it can and returns the number of bytes written.

Testing Requirements:

Test for O_NONBLOCK set by *open*() and *fcntl*().

45(PCTS_GTI_DEVICE?B:UNTESTED)

When a *write*() to a terminal returns successfully: then all of the bytes written have already been scheduled for transmission to the device.

See Reason 1 in Section 5. of POSIX.3 {4}.

7.1.1.9 Special Characters

46(PCTS_GTI_DEVICE?A:UNTESTED)

When ISIG is set and INTR is input from a terminal, then a SIGINT signal is sent to all processes in the foreground process group for which the terminal is the controlling terminal.

47(PCTS_GTI_DEVICE?A:UNTESTED)

When ISIG is set, then INTR is not placed on the input queue.

48(PCTS_GTI_DEVICE?A:UNTESTED)

When ISIG is set and the QUIT character is input from a terminal, then a SIGQUIT signal is sent to all processes in the foreground process group for which the terminal is the controlling terminal.

49(PCTS_GTI_DEVICE?A:UNTESTED)

When ISIG is set, then QUIT is not placed on the input queue.

50(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When ISIG is set and the SUSP character is input from a terminal, then a SIGTSTP signal is sent to all processes in the foreground process group for which the terminal is the controlling terminal.

51(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When ISIG is set, then SUSP is not placed on the input queue.

R05 When IXON is set, then STOP and START are not placed on the input queue. (See Assertion 31 in 7.1.2.2.)

52(PCTS_GTI_DEVICE?A:UNTESTED)

When ICANON is set, and ERASE is received, then the last character of the current line is removed from the input queue, line delimiters are not removed, and the ERASE character is discarded. Lines in the input queue are delimited by an NL or by the receipt, of an EOF or EOL character.

Testing Requirements:

Test when current line is empty and when current line contains data.

53(PCTS_GTI_DEVICE?A:UNTESTED)

When ICANON is set, and KILL is input from a terminal, then all characters in the current line are removed from the input queue, up to but not including the line delimiter, and the KILL character is discarded. Lines in the input queue are delimited by a NL or by the receipt of an EOF or EOL character.

54(PCTS_GTI_DEVICE?A:UNTESTED)

When ICANON is set, EOF is received as input data, and fewer bytes of input data are received prior to an EOF than those requested by the *read*(), then the *read*() returns the input data prior to the EOF, and the EOF character is discarded.

Testing Requirements:

Test for the conditions when the number of bytes requested is N and the bytes waiting to be read are 0 and N -1.

55(PCTS_GTI_DEVICE?A:UNTESTED)

When ICANON is set, EOF is received as input data, and more bytes of input data are received prior to an EOF than those requested by the *read*(), then the *read*() returns the input data requested, and the remaining bytes and the EOF are retained in the input stream.

Testing Requirements:

Test for the condition when the number of bytes requested is N and the bytes waiting to be read are N + 1.

56(PCTS_GTI_DEVICE?A:UNTESTED)

When ICANON is set, EOF is received as input data, and the bytes of input data received prior to an EOF is the number of bytes requested by *read*(), then the *read*() returns the input data requested.

Testing Requirements:

Test for the condition when the number of bytes requested is N and the bytes waiting to be read are N.

NOTE — An IEEE interpretation is needed to determine if the EOF is retained or deleted.

R06 When ICANON is set, and when NL (the C Standard {2} language character constant '\n') is input from a terminal, then a line is delimited. (See Assertion 23 in 7.1.1.6.)

R07 When ICANON is set and EOL is input from a terminal, then a line is delimited. (See Assertion 23 in 7.1.1.6.)

R08 When IXON is set and STOP is input from a terminal, then output to that terminal is suspended. (See Assertion 29 in 7.1.2.2.)

R09 When IXON is set, START is input from a terminal, and output to that terminal is suspended by the receipt of STOP, then output to that terminal is recommenced. (See Assertion 30 in 7.1.2.2.)

57(PCTS_GTI_DEVICE?A:UNTESTED)

The values for INTR, QUIT, ERASE, KILL, EOF, and EOI, characters can be changed.

58(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

The SUSP character can be changed.

D12(A) The details on whether the START and STOP special characters can be changed are contained in 7.1.1.9 of the PCD.1.

59(PCTS_GTI_DEVICE?C:UNTESTED)

If the changing of the START and STOP characters is supported:

The values for the START and STOP characters can be changed.

60(PCTS_GTI_DEVICE?A:UNTESTED)

When the IEXTEN flag is not set, then all characters except the special characters INTR, QUIT, SUSP, ERASE, KILL, EOF, EOL, START, and STOP are received without interpretation.

61(PCTS_GTI_DEVICE?C:UNTESTED)

If {_POSIX_VDISABLE} is in effect for the terminal file:

When the value of the special character is set to the value {_POSIX_VDISABLE}, then the action associated with that special character (INTR, QUIT, ERASE, KILL, EOF, or EOL) does not occur for any character received.

62(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported, and {_POSIX_VDISABLE} is in effect for the terminal file:

When the value of SUSP is set to the value {_POSIX_VDISABLE}, then the SUSP action does not occur for any character received.

63(PCTS_GTI_DEVICE?C :UNTESTED)

If {_POSIX_VDISABLE} is in effect for the terminal file, and the START and STOP characters can be changed:

When the values of START and STOP are set to the value {_POSIX_VDISABLE}, then the function of that character does not occur for any character received.

D13(C) If the response of the terminal interface when two or more special characters have the same value is documented:

The details of the operation performed when that character is received are contained in 7.1.1.9 of the PCD.1.

D14(C) If any single bytes other than INTR, QUIT, ERASE, KILL, EOF, NL, EOL, SUSP, STOP, START, or CR, or any multibyte character sequences, have special meaning in terminal input:

The details are contained in 7.1.1.9 of the PCD.1.

7.1.1.10 Modem Disconnect

64(PCTS_GTI_DEVICE?C:UNTESTED)

If modem control is supported:

When a modem disconnect is detected by the terminal interface for a controlling terminal, and when CLOCAL is clear, then the SIGHUP signal is sent to the controlling process associated with the terminal.

65(PCTS_GTI_DEVICE?C:UNTESTED)

If modem control is supported:

When a modem disconnect is detected, and when SIGHUP is blocked, caught, or ignored, then any subsequent *read*() to the disconnected terminal file returns zero, indicating end-of-file. This behavior continues for each subsequent call to *read*() until the device is closed.

D15(C) If both the behavior associated with {_POSIX_JOB_CONTROL} and modem control are supported, and it is documented whether an EOF condition is returned or *errno* is set to [EIO] when a modem disconnect is detected by the terminal interface:

The details are contained in 7.1.1.10 of the PCD.1.

66(PCTS_GTI_DEVICE?C:UNTESTED)

If modem control is supported:

When a modem disconnect is detected, then any subsequent *write*() to the disconnected terminal file returns a value of (*ssize_t*)-1 and sets *errno* to [EIO]. This behavior continues for each subsequent call to *write*() until the device is closed.

7.1.1.11 Closing a Terminal Device File

67(PCTS_GTI_DEVICE?A:UNTESTED)

The last process to close a terminal device file causes any data in the output queue to be sent to the device.

68(PCTS_GTI_DEVICE?B:UNTESTED)

The last process to close a terminal device file causes any data in the input queue to be discarded.

See Reason 3 in Section 5. of POSIX.3 {4}.

69(PCTS_GTI_DEVICE?C:UNTESTED)

If both modem control and HUPCL set are supported:

When the last close on the device occurs and the HUPCL flag is set, then a disconnect is performed.

7.1.2 Parameters That Can Be Set

7.1.2.1 *termios* Structure

01(A) When `<termios.h>` is included, then *struct terrnios* is declared.

D01(C) If the *termios* structure of the implementation includes members other than those specified by POSIX.1 {3}, and this is documented:

The details are contained in 7.1.2.1 of the PCD.1.

02(A) When `<termios.h>` is included, then the types *tcflag_t* and *cc_t* are defined as unsigned integral types.

03(A) When `<terminos.h>` is included, then the elements of *struct termios*: *c_iflag*, *c_oflag*, *c_cflag*, and *c_lflag* are defined, each element being of type *tcflag_t*.

04(A) When `<termios.h>` is included, then NCCS is defined as a positive integral constant.

R01 NCCS is greater than the largest special control character subscript specifically defined in POSIX.1 {3}. (See Assertion 70 in 7.1.2.6.)

05(B) NCCS is greater than or equal to the number of distinct special control character subscripts available to an implementation.

See Reason 4 in Section 5. of POSIX.3 {4}.

06(A) When `<termios.h>` is included, then *c_cc* is defined as an array having NCCS elements of type *cc_t*.

7.1.2.2 Input Modes

07(A) When `<termios.h>` is included, then the symbolic masks IGNBRK, BRKINT, IGNPAR, PARMRK, INPCK, ISTRIP, INLCR, IGNCR, ICRNL, IXON, and IXOFF are defined and are bitwise distinct from each other.

D02(A) The details of the break condition for non-asynchronous data transmission are contained in 7.1.2.2 of the PCD.1.

08(PCTS_GTI_DEVICE?A:UNTESTED)

When IGNBRK is set, then a break condition detected on input is ignored and does not affect the data read by any process.

09(PCTS_GTI_DEVICE?A:UNTESTED)

When IGNBRK is clear and BRKINT is set, then the break condition flushes the input and output queues associated with that terminal.

Testing Requirements:
Test for a controlling terminal and a noncontrolling terminal.

NOTE — When the implementation does not buffer output for the terminal, then testing for flushing of output queues does not apply.

10(PCTS_GTI_DEVICE?A:UNTESTED)

When IGNBRK is clear, BRKINT is set, and the terminal is the controlling terminal of a foreground process group, then the break condition generates a single SIGINT signal to the foreground process group.

11(PCTS_GTI_DEVICE?A:UNTESTED)

When IGNBRK and BRKINT and PARMRK are clear, then a break condition is read as a single '\0'.

12(PCTS_GTI_DEVICE?A:UNTESTED)

When IGNBRK and BRKINT are clear, and PARMRK is set, a break condition is read as '\377', '\0', '\0'.

Testing Requirements:
Test with ISTRIP set and with ISTRIP clear.

13(PCTS_GTI_DEVICE?B:UNTESTED)

When IGNPAR is set, then a byte with a framing error (other than break) is ignored.

See Reason 1 in Section 5. of POSIX.3 {4}.

14(PCTS_GTI_DEVICE?C:UNTESTED)

If PARENB set is supported:
When PARENB, INPCK, and IGNPAR are set, then a byte with a parity error is ignored.

15(PCTS_GTI_DEVICE?B:UNTESTED)

When PARMRK is set, and IGNPAR is clear, then a byte with a framing error (other than break) is read as the three-character sequence '\377', '\0', 'X', where 'X' is the data of the character received in error.

Testing Requirements:
Test with ISTRIP set and with ISTRIP clear.

See Reason 1 in Section 5. of POSIX.3 {4}.

16(PCTS_GTI_DEVICE?C:UNTESTED)

If PARENB set is supported:
When PARENB, INPCK, and PARMRK are set and IGNPAR is clear, then a byte with a parity error is read as the three-character sequence '\377', '\0', 'X', where 'X' is the data of the character received in error.
Testing Requirements:
Test with ISTRIP set and with ISTRIP clear.

17(PCTS GTI_DEVICE?C :UNTESTED)

If both PARENB set and a CSIZE of CS8 are supported:
When PARENB and PARMRK are set, when CSIZE is set to CS8, and when IGNPAR and ISTRIP are clear, then a valid character of '\377' is read as '\377', '\377'.

18(PCTS_GTI_DEVICE?B:UNTESTED)

When IGNPAR and PARMRK are clear, then a framing error (other than break) is read as single character '\0'.

See Reason 1 in Section 5. of POSIX.3 {4}.

19(PCTS_GTI_DEVICE?C:UNTESTED)

If PARENB set is supported:

When PARENB and INPCK are set, and IGNPAR and PARMRK are clear, then a parity error is given to the application as a single character '\0'.

20(PCTS_GTI_DEVICE?C:UNTESTED)

If PARENB set is supported:

When PARENB and INPCK are set, then input parity checking is enabled, and characters received with correct parity are as sent and do not generate an error.

21(PCTS_GTI_DEVICE?C:UNTESTED)

If PARENB set is supported:

When PARENB is set and INPCK is clear, then input parity checking is disabled but output parity generation will occur.

22(PCTS_GTI_DEVICE?C:UNTESTED)

If a CSIZE of CS8 is supported:

When CSIZE is set to CS8 and ISTRIP is set, then valid input characters are stripped to 7 least significant bits before returning them to the application.

23(PCTS_GTI_DEVICE?C:UNTESTED)

If a CSIZE of CS8 is supported:

When CSIZE is set to CS8 and ISTRIP is clear, then all 8 b of valid input characters are returned to the application.

24(PCTS_GTI_DEVICE?A:UNTESTED)

When INLCR is set, then a received NL character is translated into a CR character.

25(PCTS_GTI_DEVICE?A:UNTESTED)

When INLCR is clear, then a received NL character is placed on the input queue unaltered.

26(PCTS_GTI_DEVICE?A:UNTESTED)

When IGNCR is set, then a received CR character is not placed in the input queue.

27(PCTS_GTI_DEVICE?A:UNTESTED)

When IGNCR is clear and ICRNL is set, then a received CR character is translated into a NL character.

28(PCTS_GTI_DEVICE?A:UNTESTED)

When IGNCR and ICRNL are both clear, then a received CR character is not altered during input processing.

29(PCTS_GTI_DEVICE?A:UNTESTED)

When IXON is set, then a received STOP character suspends output to the terminal.

30(PCTS_GTI_DEVICE?A:UNTESTED)

When IXON is set, then a received START character restarts output to the terminal suspended by a previous STOP character.

31(PCTS_GTI_DEVICE?A:UNTESTED)

When IXON is set, then no START or STOP characters are placed in the input queue.

32(PCTS_GTI_DEVICE?A:UNTESTED)

When IXON is clear, then the START and STOP characters are placed in the input queue.

33(PCTS_GTI_DEVICE && {MAX_INPUT} ≤ {PCTS_MAX_INPUT}?A:UNTESTED)

When IXOFF is set, then the system transmits one or more STOP characters intended to prevent the input queue from overflowing.

34(PCTS_GTI_DEVICE && {MAX_INPUT} ≤ {PCTS_MAX_INPUT}?A:UNTESTED)

When IXOFF is set, when the system has stopped input from the terminal device by sending one or more STOP characters, and when sufficient data has been read from the input queue to allow input to be restarted, then the system transmits one or more START characters as needed to cause the terminal device to resume transmitting data.

D03(A) The details of the precise conditions under which STOP and START characters are transmitted to prevent the number of bytes in the input queue from exceeding {MAX_INPUT} are contained in 7.1.2.2 of the PCD.1.

D04(A) The initial input control value after *open*() is specified in 7.1.2.2 of the PCD.1.

7.1.2.3 Output Modes

35(A) When `<termios.h>` is included, then the mask OPOST is defined.

D05(A) The details describing the manner in which output data is processed when OPOST is set to modify the appearance of lines of text on the terminal device are contained in 7.1.2.3 of the PCD.1.

36(PCTS_GTI_DEVICE?A:UNTESTED)

When OPOST is clear, then characters are transmitted to the terminal without change.

D06(A) The initial output control value after *open*() is specified in 7.1.2.3 of the PCD.1.

7.1.2.4 Control Modes

D07(C) If hardware control modes are ignored for an object that is not an asynchronous serial connection, and this is documented:

The details are contained in 7.1.2.4 of the PCD.1.

37(A) When `<termios.h>` is included, then the symbolic flags CLOCAL, CREAD, CSIZE, CSTOPB, HUPCL, PARENB, and PARODD are defined and are bitwise distinct from each other.

38(A) When `<termios.h>` is included, then the CS5, CS6, CS7, and CS8 bit masks are defined to have distinct values. CSIZE is defined as a mask that includes all bits in the bitwise inclusive OR of those values.

39(PCTS_GTI_DEVICE?C:UNTESTED)

If CSIZE of CS5 is supported:

When CSIZE bits are set to CS5, then characters are sent and received as 5 b, not including the parity bit.

40(PCTS_GTI_DEVICE?C:UNTESTED)

If a CSIZE of CS6 is supported:

When CSIZE bits are set to CS6, then characters are sent and received as 6 b, not including the parity bit.

41(PCTS_GTI_DEVICE?C:UNTESTED)

If a CSIZE of CS7 is supported:

When CSIZE bits are set to CS7, then characters are sent and received as 7 b, not including the parity bit.

42(PCTS_GTI_DEVICE?C:UNTESTED)

If a CSIZE of CS8 is supported:
When CSIZE bits are set to CS8, then characters are sent and received as 8 b, not including the parity bit.

43(PCTS_GTI_DEVICE?D:UNTESTED)

If CSTOPB set is supported:
When CSTOPB is set, then two stop bits are used.

See Reason 1 in Section 5. of POSIX.3 {4}.

44(PCTS_GTI_DEVICE?D:UNTESTED)

If CSTOPB clear is supported:
When CSTOPB is clear, then one stop bit is used.

See Reason 1 in Section 5. of POSIX.3 {4}.

45(PCTS_GTI_DEVICE?C:UNTESTED)

If CSTOPB set is supported:
When CSTOPB is set, then data can be written and read.

46(PCTS_GTI_DEVICE?C:UNTESTED)

If CSTOPB clear is supported:
When CSTOPB is clear, then data can be written and read.

47(PCTS_GTI_DEVICE?C:UNTESTED)

If CREAD set is supported:
When CREAD is set, then characters can be received.

48(PCTS_GTI_DEVICE?C:UNTESTED)

If CREAD clear is supported:
When CREAD is clear, then characters cannot be received.

R02 When PARENB is set, then parity generation and detection is enabled. (See Assertions 49-52 in 7.1.2.4.)

49(PCTS_GTI_DEVICE?D:UNTESTED)

If both PARENB set and PARODD set are supported:
When PARODD is set, then parity generation uses odd parity.

See Reason 1 in Section 5. of POSIX.3 {4}.

50(PCTS_GTI_DEVICE?C:UNTESTED)

If both PARENB set and PARODD set are supported:
When PARENB and PARODD are set, then parity generation and detection is enabled.
Testing Requirements:
Test that characters received with odd parity do not generate an error and that characters received with even parity generate an error.

51(PCTS_GTI_DEVICE?D:UNTESTED)

If both PARENB set and PARODD clear are supported:
When PARODD is clear, then parity generation uses even parity.

See Reason 1 in Section 5. of POSIX.3 {4}.

52(PCTS_GTI_DEVICE?C:UNTESTED)

If both PARENB set and PARODD clear are supported:

When PARENB is set and PARODD is clear, then parity generation and detection is enabled.
Testing Requirements:
Test that characters received with even parity do not generate an error and that characters received with odd parity generate an error.

53(PCTS_GTI_DEVICE?C:UNTESTED)

If both modem control and HUPCL set are supported:
When HUPCL is set, CLOCAL is clear, and the last process with the port open closes the port or terminates, then modem disconnect occurs.

54(PCTS_GTI_DEVICE?C:UNTESTED)

If both modem control and HUPCL clear are supported:
When both HUPCL and CLOCAL are clear, and the last process with the port open closes the port or terminates, then modem disconnect does not occur.

55(PCTS_GTI_DEVICE?C:UNTESTED)

If CLOCAL set is supported:
When CLOCAL is set, then a connection is independent of the state of the modem status lines. The SIGHUP signal is not sent to the controlling process when the modem control line is lowered.

R03 When CLOCAL is clear, then a connection is dependent on the state of the modem status lines. The SIGHUP signal is sent to the controlling process when a modem disconnect is detected. (See Assertion 64 in 7.1.1.10.)

56(PCTS_GTI_DEVICE?C:UNTESTED)

For *funct*() of *open*() and *creat*():
If modem control is supported:
When CLOCAL is clear for the terminal, then a call to *funct*()—with the O_NONBLOCK flag clear in the case where *funct*() is *open*()—waits for the modem connection tc complete before returning.

57(PCTS_GTI_DEVICE?C:UNTESTED)

If modem control is supported:
When CLOCAL is clear for the terminal, then a call to *open*() with the O_NONBLOCK flag set returns without waiting for the modem connection.

58(PCTS_GTI_DEVICE?C:UNTESTED)

For *funct*() of *open*() and *creat*():
If CLOCAL set is supported:
When CLOCAL is set for the terminal, then a call to *funct*() returns without waiting for the modem connection.
Testing Requirements:
Test *open*() with O_NONBLOCK set and O_NONBLOCK clear.

D08(A) The initial hardware control value after *open*() is specified in 7.1.2.4 of the PCD.1.

7.1.2.5 Local Modes

59(A) When `<termios.h>` is included, then the symbols ECHO, ECHOE, ECHOK, ECHONL, ICANON, IEXTEN, ISIG, NOFLSH, and TOSTOP are defined and are bitwise distinct from each other.

60(PCTS_GTI_DEVICE?A:UNTESTED)

When ECHO is set, then input characters are echoed back to the terminal.

61(PCTS_GTI_DEVICE?A:UNTESTED)

When ECHO is clear, then input characters are not echoed back to the terminal.

D09(C) If the condition when ECHOE and ICANON are set and an ERASE character is encountered and there is no character to erase, and this is documented:

The details on whether the implementation echos an indication that there was no character to erase or does nothing are contained in 7.1.2.5 of the PCD.1.

62(PCTS_GTI_DEVICE?A: UNTESTED)

When ICANON and ECHOE are set, when there are characters on the current line, and when the ERASE character is received, then the {PCTS_ECHOE} string is sent to the terminal.

Testing Requirements:

Test only for ECHO set.
The PCTS.1 must state the character sequences that are expected to be erased and allow the implementation to provide different sequences for PCTS_ECHOE and PCTS_ECHOK to erase the character sequences for the default terminal type.

D10(C) If the behavior of the terminal interface when ECHOK and ICANON are set is documented:

The details as to whether the terminal causes the terminal to erase the line from the display or echoes the KILL character followed by the '\n' character are contained in 7.1.2.5 of the PCD.1.

63(PCTS_GTI_DEVICE?A:UNTESTED)

When ICANON and ECHOK are set and the KILL character is received, then the {PCTS_ECHOK} string is sent to the terminal.

Testing Requirements:

Test only for ECHO set.
The PCTS.1 must state the character sequences that are expected to be erased and allow the implementation to provide different sequences for PCTS_ECHOE and PCTS_ECHOK to erase the character sequences for the default terminal type.

64(PCTS_GTI_DEVICE?A:UNTESTED)

When ICANON and ECHONL are set, then the NL character is echoed when received.

Testing Requirements:

Test with ECHO set and with ECHO clear.

65(PCTS_GTI_DEVICE?A:UNTESTED)

When ISIG is clear, and INTR or QUIT is received, then no special character functions occur, and the characters are placed on the input queue.

66(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When ISIG is clear and SUSP is received, then no special character function occurs, and the SUSP character is placed on the input queue.

R04 When ICANON is set, then canonical mode input processing is enabled. (See Assertions 23-26 in 7.1.1.6.)

R05 When ICANON is clear, then read requests are satisfied directly from the input queue in accordance with noncanonical mode input processing. (See Assertions 27 and 28 in 7.1.1.7.)

D11(C) If any functions are recognized from input data when IEXTEN is set:

These functions are stated in 7.1.2.5 of the PCD.1.

D12(A) The interaction of IEXTEN being set with ICANON, ISIG, IXON, or IXOFF is described in 7.1.2.5 of the PCD.1.

67(PCTS_GTI_DEVICE?A:UNTESTED)

When NOFLSH and ISIG are set, and INTR or QUIT is received, then data in the input and output queues is not discarded.

NOTE — INTR and QUIT are discarded when processed.

68(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When NOFLSH and ISIG are set, and SUSP is received, then data in the input and output queues is not discarded.

NOTE — SUSP is discarded when processed.

R06 When TOSTOP is set, then TOSTOP signalling is enabled for a process writing to the controlling terminal. (See Assertions 16–18 in 7.1.1.4.)

D13(A) The initial local control value after *open*() is specified in 7.1.2.5 of the PCD.1.

7.1.2.6 Special Control Characters

69(A) When `<termios.h>` is included, then the subscripts VEOF, VEOL, VERASE, VINTR, VKILL, VMIN, VQUIT, VSUSP, VTIME, VSTART, and VSTOP are defined.

70(A) VERASE, VINTR, VKILL, VQUIT, VSUSP, VSTART, and VSTOP are all distinct and less than NCCS. VMIN and VEOF are distinct from all of the above (but not necessarily from each other) and are less than NCCS. VTIME and VEOL are distinct from all of the above (including VMIN and VEOF) (but not necessarily from each other) and are less than NCCS.

D14(C) If the implementation does not support the behavior associated with {_POSIX_JOB_CONTROL} and it is documented whether the implementation ignores the character value in the *c_cc* array indexed by the VSUSP subscript:

The details are contained in 7.1.2.6 of the PCD.1.

D15(C) If the value of NCCS (the number of elements in the *c_cc* array is documented:

The details are contained in 7.1.2.6 of the PCD.1.

D16(C) If the implementation does not support changing the START and STOP characters and it is documented whether the character values in the *c_cc* array indexed by START and STOP are ignored:

The details are contained in 7.1.2.6 of the PCD.1.

71(PCTS_GTI_DEVICE?A:UNTESTED)

A call to *tcgetattr*() returns the values for START and STOP in the appropriate elements of the *c_cc* array of the *struct termios.*

R07 The function of the special characters INTR, QUIT, ERASE, KILL, EOF, EOL, and SUSP are disabled individually by setting the value of the special character to {_POSIX_VDISABLE}. (See Assertions 61 and 62 in 7.1.1.9.)

D17(A) The initial values of all control characters are stated in 7.1.2.6 of the PCD.1.

7.1.2.7 Baud Rate Values

There are no assertions specific to this subclause.

7.1.3 Baud Rate Functions

7.1.3.1 *cfgetospeed*() (7.1.3)

7.1.3.1.1 Synopsis (7.1.3.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<termios.h>` is included, then the function prototype `speed_t cfgetospeed(const struct termios *)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header <termios.h> is included, then the function *cfgetospeed*() is declared with the result type *speed_t*. (See GA36 in 2.7.3.)

02(C) If *cfgetospeed*() is defined as a macro when the header <termios.h> is included:

When the macro cfgetospeed() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *speed_t*. (See GA37 in 2.7.3.)

03(C) If *cfgetospeed*() is defined as a macro in the header <termios.h>:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

7.1.3.1.2 Description (7.1.3.2)

04(A) When <termios.h> is included, then the symbolic baud rates B0, B50, B75, B110, B134, B150, B200, B300, B600, B1200, B1800, B2400, B4800, B9600, B19200, and B38400 are defined.

05(A) The type *speed_t* is defined in <termios.h> and is an unsigned integral type.

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

7.1.3.1.3 Returns (7.1.3.3)

There are no assertions specific to this subclause.

7.1.3.1.4 Errors (7.1.3.4)

D01(C) If error conditions are detected for *cfgetospeed*(), and this is documented:

The details on the error conditions detected are contained in 7.1.3.4 of the PCD.1.

7.1.3.2 *cfsetospeed*() (7.1.3)

7.1.3.2.1 Synopsis (7.1.3.1)

01(A) If the implementation provides C Standard {2} support:

When the header <termios.h> is included, then the function prototype `int cfsetospeed(struct termios *, speed_t)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header <termios.h> is included, then the function *cfsetospeed*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *cfsetospeed*() is defined as a macro when the header <termios.h> is included:

When the macro *cfsetospeed*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type int. (See GA37 in 2.7.3.)

03(C) If *cfsetospeed*() is defined as a macro in the header <termios.h>:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

7.1.3.2.2 Description (7.1.3.2)

04(A) A call to *cfsetospeed*(*termios_p*, *speed*) sets the output baud rate stored in the *termios* structure pointed to by the *termios_p* parameter to *speed*.

05(PCTS_GTI_DEVICE?A:UNTESTED)

For each of the following baud rates that are supported: B50, B75, B110, B134, B150, B200, B300, B600, B1200, B1800, B2400, B4800, B9600, B19200, and B38400, a call to *cfsetospeed*() that returns 0, followed by a *tcsetattr*() that returns 0, sets the output baud rate to the corresponding speed. At least one baud rate must be supported.

D01(C) If it is documented whether *cfsetospeed*() returns an error when an attempt is made to set an unsupported baud rate:

The details are contained in 7.1.3.2 and 7.1.3.4 of the PCD.1.

7.1.3.2.3 Returns (7.1.3.3)

There are no assertions specific to this subclause.

7.1.3.2.4 Errors (7.1.3.4)

D02(C) If error conditions are detected for *cfsetospeed*(), and this is documented:

The details on the error conditions detected are contained in 7.1.3.4 of the PCD.1.

7.1.3.3 *cfgetispeed*() (7.1.3)

7.1.3.3.1 Synopsis (7.1.3.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<termios.h>` is included, then the function prototype `speed_t cfgetispeed(const struct termios *)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<termios.h>` is included, then the function *cfgetispeed*() is declared with the result type *speed_t*. (See GA36 in 2.7.3.)

02(C) If *cfgetispeed*() is defined as a macro when the header `<termios.h>` is included:

When the macro *cfgetispeed*() is invoked with the correct argument types (or compatible argument types in the case, that C Standard {2} support is provided), then it expands to an expression with the result type *speed_t*. (See GA37 in 2.7.3.)

03(C) If *cfgetispeed*() is defined as a macro in the header `<termios.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

7.1.3.3.2 Description (7.1.3.2)

04(A) A call to *cfgetispeed*(*termios_p*) returns the input baud rate stored in the termios structure pointed to by the *termios_p* parameter.

7.1.3.3.3 Returns (7.1.3.3)

There are no assertions specific to this subclause.

7.1.3.3.4 Errors (7.1.3.4)

D01(C) If error conditions are detected for *cfgetispeed*(), and this is documented:

The details on the error conditions detected are contained in 7.1.3.4 of the PCD.1.

7.1.3.4 *cfsetispeed*() (7.1.3)

7.1.3.4.1 Synopsis (7.1.3.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<termios.h>` is included, then the function prototype `int cfsetispeed(struct termios *, speed_t)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<termios.h>` is included, then the function *cfsetispeed*() is either declared with result type int or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *cfsetispeed*() is defined as a macro when the header `<termios.h>` is included:

When the macro *cfsetispeed*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *cfsetispeed*() is defined as a macro in the header `<termios.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

7.1.3.4.2 Description (7.1.3.2)

04(A) A call to *cfsetispeed*(*termios_p*, *speed*) sets the input baud rate stored in the *termios* structure pointed to by the *termios_p* parameter to *speed*.

05(PCTS_GTI_DEVICE?A:UNTESTED)

For each of the following baud rates that are supported: B50, B75, B110, B134, B150, B200, B300, B600, B1200, B1800 B2400, B4800, B9600, B19200, and B38400, a call to *cfsetispeed*() that returns 0, followed by a call to *tcsetattr*() that returns 0, sets the input baud rate to the corresponding speed. At least one baud rate must be supported.

D01(C) If it is documented whether *cfsetispeed*() returns an error when an attempt is made to set an unsupported baud rate:

The details are contained in 7.1.3.2 and 7.1.3.4 of the PCD.1.

7.1.3.4.3 Returns (7.1.3.3)

There are no assertions specific to this subclause.

7.1.3.4.4 Errors (7.1.3.4)

D02(C) If error conditions are detected for *cfsetispeed*(), and this is documented:

The details on the error conditions detected are contained in 7.1.3.4 of the PCD.1.

7.2 General Terminal Interface Control Functions

GA60 For *funct*() of *tcsetattr*(), *tcsendbreak*(), *tcdrain*(), *tcflow*(), and *tcflush*():

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

A call to *funct*() on the controlling terminal of the process from a member of a background process group that is not blocking or ignoring SIGTTOU signals causes a SIGTTOU signal to be sent to its process group. The operation of *funct*()is not performed prior to the signal being received.

Testing Requirements:

Test when TOSTOP is set and when TOSTOP is clear.

GA61 For *funct*() of *tcsetattr*(), *tcsendbreak*(), *tcdrain*(), *tcflow*(), and *tcflush*():
If the behavior associated with {_POSIX_JOB_CONTROL} is supported:
When a member of a background process group that is blocking SIGTTOU signals calls *funct*() on the controlling terminal of the process, then *funct*() is allowed to perform the actions and returns the value zero, and the SIGTTOU signal is not sent to its process group.
Testing Requirements:
Test when TOSTOP is set and when TOSTOP is clear.

GA62 For *funct*() of *tcsetattr*(), *tcsendbreak*(), *tcdrain*(), *tcflow*(), and *tcflush*():
If the behavior associated with {_POSIX_JOB_CONTROL} is supported:
When a member of a background process group that is ignoring SIGTTOU signals calls *funct*() on the controlling terminal of the process, then *funct*() is allowed to perform the actions and returns the value zero, and the SIGTTOU signal is not sent to its process group.
Testing Requirements:
Test when TOSTOP is set and when TOSTOP is clear.

There are no assertions specific to this subclause.

7.2.1 Get and Set State

7.2.1.1 *tcgetattr*() (7.2.1)

7.2.1.1.1 Synopsis (7.2.1.1)

01(A) If the implementation provides C Standard {2} support:
When the header `<termios.h>` is included, then the function prototype
`int tcgetattr(int, struct termios *)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<termios.h>` is included, then the function *tcgetattr*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *tcgetattr*() is defined as a macro when the header `<termios.h>` is included:
When the macro *tcgetattr*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *tcgetattr*() is defined as a macro in the header `<termios.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

7.2.1.1.2 Description (7.2.1.2)

04(PCTS_GTI_DEVICE?A:UNTESTED)

A call to *tcgetattr*(*fildes*, *termios_p*) stores the parameters associated with the terminal in the termios structure referenced by *termios_p* and returns zero.

Testing Requirements:
If {_POSIX_JOB_CONTROL} is supported, test the behavior for a process that is a member of a foreground process group and for a process that is a member of a background process group.

D01(C) If support or nonsupport for input and output baud rates that differ is documented:
The details are contained in 7.2.1.2 of the PCD.1.

05(PCTS_GTI_DEVICE?C:UNTESTED)

If the implementation supports input and output baud rates that differ:

The baud rates stored in the *termios* structure returned by *tcgetattr*() reflect the actual baud rates even when they are equal.

06(PCTS_GTI_DEVICE?C:UNTESTED)

If the implementation does not support input and output baud rates that differ:

The output baud rate stored in the *termios* structure returned by *tcgetattr*() is the actual baud rate. The input baud rate stored in this *termios* structure is either equal to zero or equal to the actual baud rate.

7.2.1.1.3 Returns (7.2.1.3)

R01 When a call to *tcgetattr*() completes successfully, then a value of (*int*)0 is returned. (See Assertion 4 in 7.2.1.1.2.)

R02 When a call to *tcgetattr*() completes unsuccessfully, then a value of (*int*)−1 is returned and sets *errno* to indicate the error. (See Assertions 7 and 8 in 7.2.1.1.4.)

7.2.1.1.4 Errors (7.2.1.4)

07(A) When *fildes* is not a valid file descriptor, then a call to *tcgetattr*(*fildes*, *termios_p*) returns a value of (*int*)−1 and sets *errno* to [EBADF].

08(A) When the file associated with *fildes* is not a terminal, then a call to *tcgetattr*(*fildes*, *termios_p*) returns a value of (*int*)−1 and sets *errno* to [ENOTTY].

7.2.1.2 *tcsetattr*() (7.2.1)

7.2.1.2.1 Synopsis (7.2.1.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<termios.h>` is included, then the function prototype `int tcsetattr(int, int, const struct termios *)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<termios.h>` is included, then the function *tcsetattr*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *tcsetattr*() is defined as a macro when the header `<termios.h>` is included:

When the macro tcsetattr() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *tcsetattr*() is defined as a macro in the header `<termios.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

7.2.1.2.2 Description (7.2.1.2)

04(A) When the header `<termios.h>` is included, then the symbols TCSANOW, TCSADRAIN, and TCSAFLUSH are defined and are different from each other.

05(PCTS_GTI_DEVICE?A:UNTESTED)

A call to *tcsetattr*(*fildes*, *TCSANOW*, *termios_p*) from a foreground process immediately sets the parameters associated with the terminal from the *termios* structure referenced by *termios_p*, without waiting for data to drain or to be flushed, and returns the value zero.

06(PCTS_GTI_DEVICE?B:UNTESTED)

A call to *tcsetattr*(*fildes, optional_action, termios_p*) from a foreground process sets the parameters associated with the terminal from the *termios* structure referenced by *termios_p* after the specified *optional_action* has been performed.

Testing Requirements:

Test for *optional_action* of TCSADRAIN and TCSAFLUSH.

See Reason 2 in Section 5. of POSIX.3 {4}.

07(PCTS_GTI_DEVICE?A:UNTESTED)

When a call to *tcsetattr*(*fildes, TCSADRAIN, termios_p*) is made from a foreground process, then the process outputs all data written to the terminal before returning the value zero.

08(PCTS_GTI_DEVICE?A:UNTESTED)

When a call to *tcsetattr*(*fildes, TCSAFLUSH, termios_p*) is made from a foreground process, then the process outputs all data written to the terminal and discards all input that has been received but not read before returning the value zero.

09(PCTS_GTI_DEVICE?B:UNTESTED)

A call to *tcsetattr*(*fildes, TCSAFLUSH, termios_p*) from a foreground process sets the parameters associated with the terminal from the *termios* structure referenced by *termios_p* after all output written to the terminal has been transmitted and all input that has been received but not read is discarded, and returns the value zero.

See Reason 2 in Section 5. of POSIX.3 {4}.

10(PCTS_GTI_DEVICE?C:UNTESTED)

If modem control is supported:

When the output speed in the *termios* structure is set to B0 and *tcsetattr*() is called, then the modem control lines are no longer asserted.

11(PCTS_GTI_DEVICE?A:UNTESTED)

When the input baud rate in the *termios* structure is set to zero and *tcsetattr*() is called, then the input baud rate is changed to the value of the output baud rate as specified in the *termios* structure.

12(PCTS_GTI_DEVICE?C:UNTESTED)

If the implementation does not support an attribute represented in the *termios* structure:

When an attempt is made to change an attribute represented in the *termios* structure to an unsupported value, and when another attribute is changed to a supported value, then a call to *tcsetattr*() returns zero, sets all attributes that the implementation supports as requested, and leaves unchanged any attributes that are not supported.

13(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

A call to *tcsetattr*(), on the controlling terminal of the process, from a member of a background process group that is not blocking or ignoring SIGTTOU signals causes a SIGTTOU signal to be sent to its process group. The terminal attributes are left unchanged, and any flush or drain action specified by *optional_actions* is not performed. (See GA60 in 7.2.)

14(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When a member of a background process group that is blocking SIGTTOU signals calls *tcsetattr*() on the controlling terminal of the process, then *tcsetattr*() is allowed to perform the actions and returns the value zero, and the SIGTTOU signal is not sent to its process group. (See GA61 in 7.2.)

15(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When a member of a background process group that is ignoring SIGTTOU signals calls *tcsetattr*() on the controlling terminal of the process, then *tcsetattr*() is allowed to perform the actions and returns the value zero, and the SIGTTOU signal is not sent to its process group. (See GA62 in 7.2.)

16(PCTS_GTI_DEVICE?A:UNTESTED)

When fildes refers to a noncontrolling terminal, then a call to *tcsetattr*(*fildes*, *optional_actions*, *termios_p*) sets the parameters associated with the terminal from the *termios* structure referenced by *termios_p*, carries out the actions indicated by *optional_actions*, and returns a value of zero.

Testing Requirements:

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

Test from a foreground process, a background process that is neither blocking nor ignoring SIGTTOU, a background process that is blocking SIGTTOU, a background process that is ignoring SIGTTOU, and a process that is a member of an orphaned background process group and that is neither blocking nor ignoring SIGTTOU.

7.2.1.2.3 Returns (7.2.1.3)

R01 When a call to *tcsetattr*() completes successfully, then a value of (*int*)0 is returned. (See Assertions 5, 6, 11, 12, and 16 in 7.2.1.2.2.)

R02 When a call to *tcsetattr*() completes unsuccessfully, then a value of (*int*)–1 is returned and sets *errno* to indicate the error. (See Assertions 17-21 in 7.2.1.2.4.)

7.2.1.2.4 Errors (7.2.1.4)

17(PCTS_GTI_DEVICE?A:UNTESTED)

When *fildes* is not a valid file descriptor, then a call to *tcsetattr*(*fildes*, *optional_actions*, *termios_p*) returns a value of (*int*)–1 and sets *errno* to [EBADF]. The values in the *termios* structure are not changed.

18(PCTS_GTI_DEVICE && {_EPOSIX_JOB_CONTROL}?A:UNTESTED)

When a call to *tcsetattr*() is interrupted by a signal, then it returns a value of [(*int*)-1] and *errno* is set to [EINTR].

Testing Requirements:

When the behavior associated with {_POSIX_JOB_CONTROL} is supported, test by calling *tcsetattr*() from a member of a background process group that is catching and not blocking SIGTTOU signals. Also test that the values in the termios structure are not changed, the terminal attributes are left unchanged, and any flush or drain action specified by *optional_actions* is not performed.

See Reason 1 in Section 5. of POSIX.3 {4}.

19(PCTS_GTI_DEVICE?A:UNTESTED)

When *optional_actions* has an invalid value, then a call to *tcsetattr*(*fildes*, *optional_actions*, *termios_p*) returns a value of (*int*)-1 and sets *errno* to [EINVAL]. The values in the *termios* structure are not changed, and the terminal atributes are left unchanged.

20(PCTS_GTI_DEVICE?C:UNTESTED)

If the implementation does not support an attribute represented in the *termios* structure:

When an attempt is made to change an attribute represented in the termios structure to an unsupported value, and when no other attribute is changed to a supported value, then a call to *tcsetattr*(*fildes*, *optional_actions*, *termios_p*) returns a value of (*int*)-1 and sets *errno* to [EINVAL].

The values in the *termios* structure and the terminal attributes are left unchanged, and any flush or drain action specified by *optional_actions* is not performed.

21(PCTS_GTI_DEVICE?A:UNTESTED)

When the file associated with *fildes* is not a terminal, then a call *tcsetattr*(*fildes, optional_actions, termios_p*) returns a value of (*int*)-1 and sets *errno* to [ENOTTY]. The values in the *termios* structure are not changed.

7.2.2 Line Control Functions

7.2.2.1 *tcsendbreak* () (7.2.2)

7.2.2.1.1 Synopsis (7.2.2.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<termios.h>` is included, then the function prototype `int tcsendbreak(int, int)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<termios.h>` is included, then the function *tcsendbreak*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *tcsendbreak*() is defined as a macro when the header `<termios.h>` is included:

When the macro *tcsendbreak*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *tcsendbreak*() is defined as a macro in the header `<termios.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

7.2.2.1.2 Description (7.2.2.2)

04(PCTS_GTI_DEVICE?A:UNTESTED)

A call to *tcsendbreak*(*fildes, 0*), on the controlling terminal of the process, causes transmission of a continuous stream of zero-valued bits for at least 0,25 s, and not more than 0,5 s, and returns the value zero.

D01(A) The period of time the break signal is sent when *tcsendbreak*() is called with a nonzero value for *duration* is specified in 7.2.2.2 of the PCD.1.

D02(C) If non-asynchronous serial data transmission with the terminal is supported:

The details on whether the *tcsendbreak*() function sends data to generate a break condition or returns without any action are contained in 7.2.2.2 of the PCD.1.

05(PCTS_GTI_DEVICE?B:UNTESTED)

A call to *tcsendbreak*(*fildes, duration*) sends a continuous stream of zero-valued bits for an implementation-defined period of time, and the return value is zero.

See Reason 2 in Section 5. of POSIX.3 {4}.

06(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

A call to *tcsendbreak*(), on the controlling terminal of the process, from a member of a background process group that is not blocking or ignoring SIGTTOU signals causes a SIGTTOU signal to be sent to its process group. The transmission of zero-valued bits is not performed. (See GA60 in 7.2.)

07(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When a member of a background process group that is blocking SIGTTOU signals calls *tcsendbreak*() on the controlling terminal of the process, the *tcsendbreak*() is allowed to perform the actions and returns the value zero, and the SIGTTOU signal is not sent to its process group. (See GA61 in 7.2.)

08(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When a member of a background process group that is ignoring SIGTTOU signals calls *tcsendbreak*() on the controlling terminal of the process, then *tcsendbreak*() is allowed to perform the actions and returns the value zero, and the SIGTTOU signal is not sent to its process group. (See GA62 in 7.2.)

09(PCTS_GTI_DEVICE?A:UNTESTED)

When *fildes* refers to a noncontrolling terminal, then a call to *tcsendbreak*(*fildes*, *O*) causes transmission of a break signal and returns the value zero.

Testing Requirements:

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

Test from a foreground process, a background process that is neither blocking nor ignoring SIGTTOU, a background process that is blocking SIGTTOU, a background process that is ignoring SIGTTOU, and a process that is a member of an orphaned background process group and that is neither blocking nor ignoring SIGTTOU.

7.2.2.1.3 Returns (7.2.2.3)

R01 When a call to *tcsendbreak*() completes successfully, then a value of (*int*)0 is returned. (See Assertions 4, 5, 7, and 8 in 7.2.2.1.2.)

R02 When a call to *tcsendbreak*() completes unsuccessfully, then a value of (*int*)−1 is returned and sets *errno* to indicate the error. (See Assertions 10 and 11 in 7.2.2.1.4.)

7.2.2.1.4 Errors (7.2.2.4)

10(A) When *fildes* is not a valid file descriptor, then a call to *tcsendbreak*(*fildes*, *duration*) returns a value of (*int*)−1 and sets *errno* to [EBADF].

11(A) When the file associated with *fildes* is not a terminal, then a call to *tcsendbreak*(*fildes*, *duration*) returns a value of (*int*)−1 and sets *errno* to [ENOTTY].

7.2.2.2 *tcdrain*() (7.2.2)

7.2.2.2.1 Synopsis (7.2.2.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<termios.h>` is included, then the function prototype `int tcdrain(int)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<termios.h>` is included, then the function *tcdrain*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *tcdrain*() is defined as a macro when the header `<termios.h>` is included:

When the macro *tcdrain*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *tcdrain*() is defined as a macro in the header `<termios.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4)

7.2.2.2.2 Description (7.2.2.2)

04(PCTS_GTI__DEVICE?A:UNTESTED)

A call to *tcdrain*(), on the controlling terminal of the process, returns after all output written to the terminal has been transmitted and returns zero on successful completion.

05(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

A call to *tcdrain*(), on the controlling terminal of the process, from a member of a background process group that is not blocking or ignoring SIGTTOU signals causes a SIGTTOU signal to be sent to its process group. The drain operation is not performed. (See GA60 in 7.2.)

NOTE — When the implementation does not buffer output for the terminal, then the call to tcdrain() only causes a SIGTTOU signal to be sent to its process group.

06(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When a member of a background process group that is ignoring SIGTTOU signals calls *tcdrain*() on the controlling terminal of the process, then *tcdrain*() is allowed to perform the actions and returns the value zero, and the SIGTTOU signal is not sent to its process group. (See GA61 in 7.2.)

07(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When a member of a background process group that is blocking SIGTTOU signals calls *tcdrain*() on the controlling terminal of the process, then *tcdrain*() is allowed to perform the actions and returns the value zero, and the SIGTTOU signal is not sent to its process group. (See GA62 in 7.2.)

08(PCTS_GTI_DEVICE?A:UNTESTED)

When *fildes* refers to a noncontrolling terminal, then a call to *tcdrain*(*fildes*) returns after all output written to the terminal has been transmitted and returns zero on successful completion.

Testing Requirements:

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

Test from a foreground process, a background process that is neither blocking nor ignoring SIGTTOU, a background process that is blocking SIGTTOU, a background process that is ignoring SIGTTOU, and a process that is a member of an orphaned background process group and that is neither blocking nor ignoring SIGTTOU.

7.2.2.2.3 Returns (7.2.2.3)

R01 When a call to *tcdrain*() completes successfully, then a value of (*int*)0 is returned. (See Assertions 4, 6, and 7 in 7.2.2.2.2.)

R02 When a call to *tcdrain*() completes unsuccessfully, then a value of (*int*)-1 is returned and sets *errno* to indicate the error (See Assertions 9–11 in 7.2.2.2.4.)

7.2.2.2.4 Errors (7.2.2.4)

09(A) When *fildes* is not a valid file descriptor, then a call to *tcdrain*(*fildes*) returns a value of (*int*)-1 and sets *errno* to [EBADF].

10(A) When the file associated with *fildes* is not a terminal, then a call to *tcdrain*(*fildes*) returns a value of (*int*)-1 and sets *errno* to [ENOTTY].

11(PCTS_GTI_DEVICE && PCTS_GTI_BUFFERS_OUTPUT?A:UNTESTED)

When a call to *tcdrain*(*fildes*) is interrupted by a signal, then *tcdrain*() returns a value of (*int*)-1 and sets *errno* to [EINTR].

7.2.2.3 *tcflow* () (7.2.2)

7.2.2.3.1 Synopsis (7.2.2.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<termios.h>` is included, then the function prototype `int tcflow(int, int)` is declared. (See GS36 in 2.7.3.)

Otherwise:

When the header `<termios.h>` is included, then the function *tcflow*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *tcflow*() is defined as a macro when the header `<termios.h>` is included:

When the macro *tcflow*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *tcflow*() is defined as a macro in the header `<termios.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

7.2.2.3.2 Description (7.2.2.2)

04(A) When the header `<termios.h>` is included, then the symbols TCOOFF, TCOON, TCIOFF, and TCION are defined and are different from each other.

05(PCTS_GTI_DEVICE?A:UNTESTED)

A call to *tcflow*(*fildes*, *TCOOFF*), on the controlling terminal of the process, suspends output, and the return value is zero.

06(PCTS_GTI_DEVICE?A:UNTESTED)

A call to *tcflow*(*fildes*, *TCOON*) restarts suspended output, and the return value is zero.

07(PCTS_GTI_DEVICE?A:UNTESTED)

A call to *tcflow*(*fildes*, *TCIOFF*) causes the system to transmit a STOP character, and the return value is zero.

Testing Requirements:

Test when data transmission on the line is suspended and not suspended.

08(PCTS_GTI_DEVICE?A:UNTESTED)

A call to *tcflow*(*fildes*, *TCION*) causes the system to transmit a START character to restart suspended input, and the return value is zero.

Testing Requirements:

Test when data transmission on the line is suspended and not suspended.

09(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

A call to *tcflow*(*fildes, action*), on the controlling terminal of the process, from a member of a background process group that is not blocking or ignoring SIGTTOU signals causes a SIGTTOU signal to be sent to its process group. The operation specified by action is not performed. (See GA60 in 7.2.)

10(PCTS_GTI__DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} are supported:

When a member of a background process group that is blocking SIGTTOU signals calls *tcflow*() on the controlling terminal of the process, then *tcflow*() is allowed to perform the actions and returns the value zero, and the SIGTTOU signal is not sent to its process group. (See GA61 in 7.2.)

11(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When a member of a background process group that is ignoring SIGTTOU signals calls *tcflow*() on the controlling terminal of the process, then *tcflow*() is allowed to perform the, actions and returns the value zero, and the SIGTTOU signal is not sent to its process group. (See GA62 in 7.2.)

12(PCTS_GTI_DEVICE?A:UNTESTED)

When the first *open*() of a terminal file is done, then neither its input nor its output are suspended.

13(PCTS_GTI_DEVICE?A:UNTESTED)

When *fildes* refers to a noncontrolling terminal, then a call to *tcflow*(*fildes, TCOOFF*) suspends output, and the return value is zero.

Testing Requirements:

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

Test from a foreground process, a background process that is neither blocking nor ignoring SIGTTOU, a background process that is blocking SIGTTOU, a background process that is ignoring SIGTTOU, and a process that is a member of an orphaned background process group and that is neither blocking nor ignoring SIGTTOU.

7.2.2.3.3 Returns (7.2.2.3)

R01 When a call to *tcflow*() completes successfully, then a value of (*int*)0 is returned. (See Assertions 5-8, 10, and 11 in 7.2.2.3.2.)

R02 When a call to *tcflow*() completes unsuccessfully, then a value of (*int*)-1 is returned and sets *errno* to indicate the error (See Assertions 14–16 in 7.2.2.3.4.)

7.2.2.3.4 Errors (7.2.2.4)

14(A) When *fildes* is not a valid file descriptor, then a call to *tcflow*(*fildes, action*) returns a value of (*int*)-1 and sets *errno* to [EBADF].

15(PCTS_GTI_DEVICE?A:UNTESTED)

When *action* does not have a proper value, then a call to *tcflow*(*fildes, action*) returns a value of (*int*)-1 and sets *errno* to [EINVAL].

16(A) When the file associated with *fildes* is not a terminal, then a call to *tcflow*(*fildes, action*) returns a value of (*int*)-1 and sets *errno* to [ENOTTY].

7.2.2.4 *tcflush* () (7.2.2)

7.2.2.4.1 Synopsis (7.2.2.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<termios.h>` is included, then the function prototype `int tcflush(int, int)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<termios.h>` is included, then the function *tcflush*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *tcflush*() is defined as a macro when the header `<termios.h>` is included:

When the macro *tcflush*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *tcflush*() is defined as a macro in the header `<termios.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

7.2.2.4.2 Description (7.2.2.2)

04(A) When the header `<termios.h>` is included, then the symbols TCIFLUSH, TCOFLUSH, and TCIOFLUSH are defined and are different from each other.

05(PCTS_GTI_DEVICE?A:UNTESTED)

A call to *tcflush*(*fildes*, *TCIFLUSH*), on the controlling terminal of the process, discards data received from the terminal but not read and returns the value zero.

06(PCTS_GTI_DEVICE?A:UNTESTED)

A call to *tcflush*(*fildes*, *TCOFLUSH*) discards data written to the terminal but not transmitted and returns the value zero.

NOTE — When the implementation does not buffer output for the terminal, then the call to *tcflush*() only returns the value zero.

07(PCTS_GTI_DEVICE?A:UNTESTED)

A call to *tcflush*(*fildes*, *TCIOFLUSH*) discards both data received from the terminal but not read and data written to the terminal but not transmitted, and returns the value zero.

NOTE — When the implementation does not buffer output for the terminal, then the call to *tcflush*() only discards data received from the terminal but not read and returns the value zero.

08(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

A call to *tcflush*(), on the controlling terminal of the process, from a member of a background process group that is not blocking or ignoring SIGTTOU signals causes a SIGTTOU signal to be sent to its process group. The flush operation is not performed. (See GA60 in 7.2.)

NOTE — When the implementation does not buffer output for the terminal, then the call to *tcflush*() only causes a SIGTTOU signal to be sent to its process group and does not discard data received from the terminal but not read.

09(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When a member of a background process group that is blocking SIGTTOU signals calls *tcflush*() on the controlling terminal of the process, then *tcflush*() is allowed to perform the actions and returns the value zero, and the SIGTTOU signal is not sent to its process group. (See GA61 in 7.2.)

NOTE — When the implementation does not buffer output for the terminal, then the call to *tcflush*() is allowed only to discard data received from the terminal but not read and to return the value zero.

10(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When a member of a background process group that is ignoring SIGTTOU signals calls *tcflush*() on the controlling terminal of the process, then *tcflush*() is allowed to perform the actions and returns the value zero, and the SIGTTOU signal is not sent to its process group. (See GA62 in 7.2.)

NOTE — When the implementation does not buffer output for the terminal, then the call to *tcflush*() is allowed only to discard data received from the terminal but not read and to return the value zero.

11(PCTS_GTI_DEVICE?A:UNTESTED)

When *fildes* refers to a noncontrolling terminal, then a call to *tcflush*(*fildes*, *TCIFLUSH*) discards data received from the terminal but not read and returns the value zero.

Testing Requirements:

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

Test from a foreground process, a background process that is neither blocking nor ignoring SIGTTOU, a background process that is blocking SIGTTOU, a background process that is ignoring SIGTTOU, and a process that is a member of an orphaned background process group and that is neither blocking nor ignoring SIGTTOU.

7.2.2.4.3 Returns (7.2.2.3)

R01 When a call to *tcflush*() completes successfully, then a value of (*int*)0 is returned. (See Assertions 5-7, 9, and 10 in 7.2.2.4.2.)

R02 When a call to *tcflush*() completes unsuccessfully, then a value of (*int*)-1 is returned and sets *errno* to indicate the error (See Assertions 12–14 in 7.2.2.4.4.)

7.2.2.4.4 Errors (7.2.2.4)

12(A) When *fildes* is not a valid file descriptor, then a call to *tcflush*(*fildes*, *queue_selector*) returns a value of (*int*)-1 and sets *errno* to [EBADF].

13(PCTS_GTI_DEVICE?A:UNTESTED)

When *queue_selector* does not have a proper value, then a call to *tcflush*(*fildes*, *queue_selector*) returns a value of (*int*)-1 and sets *errno* to [EINVAL].

14(A) When the file associated with *fildes* is not a terminal, then a call to *tcflush*(*fildes*, *queue_selector*) returns a value of (*int*)-1 and sets *errno* to [ENOTTY].

7.2.3 *tcgetpgrp*()

7.2.3.1 Synopsis

01(A) If the implementation provides C Standard {2} support:

When the header `<unistd.h>` is included, then the function prototype `pid_t tcgetpgrp(int)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<unistd.h>` is included, then the function *tcgetpgrp*() is declared with the result type *pid_t*. (See GA36 in 2.7.3.)

02(C) If *tcgetpgrp*() is defined as a macro when the header `<unistd.h>` is included:
When the macro *tcgetpgrp*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *pid_t*. (See GA37 in 2.7.3.)

03(C) If *tcgetpgrp*() is defined as a macro in the header `<unistd.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

7.2.3.2 Description

D01(C) If the behavior associated with {_POSIX_JOB_CONTROL} being set is not supported, and if it is documented whether *tcgetpgrp*() is supported:
The details are contained in 7.2.3.2 of the PCD.1.

04(PCTS_GTI_D EVICE?C:UNTESTED)

If either the behavior associated with {_POSIX_JOB_CONTROL} or *tcgetpgrp*() is supported:
A call to *tcgetpgrp*() returns the value of the process group ID of the foreground process group associated with the terminal.
Testing Requirements:
If the behavior associated with {_POSIX_JOB_CONTROL} is supported, test behavior of a process that is a member of a foreground process group and a process that is a member of a background process group.

7.2.3.3 Returns

R01 When a call to *tcgetpgrp*() completes successfully, then the process group ID of the foreground process group associated with the terminal is returned. (See Assertion 4 in 7.2.3.2.)

05(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:
When a call to *tcgetpgrp*() is made from a process that is a member of a session that has no foreground process group, then *tcgetpgrp*() returns a value greater than 1 that does not match the process group ID of any existing process.

R02 When a call to *tcgetpgrp*() completes unsuccessfully, then a value of (*pid_t*)-1 is returned and sets *errno* to indicate the error (See Assertions 6-9 in 7.2.3.4.)

7.2.3.4 Errors

06(C) If either the behavior associated with {_POSIX_JOB_CONTROL} or *tcgetpgrp*() is supported:
When *fildes* is not a valid file descriptor, then a call to *tcgetpgrp*(*fildes*) returns a value of (*pid_t*)-1 and sets *errno* to [EBADF].

07(C) If both the behavior associated with {_POSIX_JOB_CONTROL} and *tcgetpgrp*() are not supported:
A call to *tcgetpgrp*() is unsuccessful, returns a value of (*pid_t*)-1, and sets *errno* to [ENOSYS].

08(PCTS_GTI_DEVICE?C:UNTESTED)

If either the behavior associated with {_POSIX_JOB_CONTROL} or *tcgetpgrp*() is supported:
When the calling process does not have a controlling terminal, then a call to *tcgetpgrp*(*fildes*) returns a value of (*pid_t*)-1 and sets *errno* to [ENOTTY].

09(C) If either the behavior associated with {_POSIX_JOB_CONTROL} or *tcgetpgrp*() is supported:
When the file associated with *fildes* is not the controlling terminal of the calling process, then a call to *tcgetpgrp*(*fildes*) returns a value of (*pid_t*)-1 and sets *errno* to [ENOTTY].
Testing Requirements:

Test for a file that is not a terminal and, if the implementation provides device types that support the general terminal interface, for a terminal that is not the controlling terminal of the process.

7.2.4 *tcsetpgrp*()

7.2.4.1 Synopsis

01(A) If the implementation provides C Standard {2} support:

When the header `<unistd.h>` is included, then the function prototype `int tcsetpgrp(int, pid_t)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<unistd.h>` is included, then the function *tcsetpgrp*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *tcsetpgrp*() is defined as a macro when the header `<unistd.h>` is included:

When the macro *trsetpgrp*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *tcsetpgrp*() is defined as a macro in the header `<unistd.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

7.2.4.2 Description

D01(C) If the behavior associated with {_POSIX_JOB_CONTROL} being set is not supported, and if it is documented whether *tcsetpgrp*() is supported:

The details are contained in 7.2.4.2 of the PCD.1.

04(PCTS_GTI_DEVICE?C:UNTESTED)

If either the behavior associated with {_POSIX_JOB_CONTROL} or *tcsetpgrp*() is supported:

When the file associated with *fildes* is the controlling terminal of the calling process, when the controlling terminal is associated with the session of the calling process, and when *pgrp_id* is the process group ID of a process in the same session as the calling process, then a call to *tcsetpgrp*(*fildes*, *pgrp_id*) sets the foreground process group ID associated with the terminal to *pgrp_id* and returns a value of zero.

7.2.4.3 Returns

R01 When a call to *tcsetpgrp*() completes successfully, then a value of (*int*)0 is returned. (See Assertion 4 in 7.2.4.2.)

R02 When a call to *tcsetpgrp*() completes unsuccessfully, then a value of (*int*)-1 is returned and sets *errno* to indicate the error. (See Assertions 5–11 in 7.2.4.4.)

7.2.4.4 Errors

05(C) If either the behavior associated with {_POSIX_JOB_CONTROL} or *tcsetpgrp*() is supported:

When *fildes* is not a valid file descriptor, then a call to *tcsetpgrp*(*fildes*, *pgrp_id*) returns a value of (*int*)-1 and sets *errno* to [EBADF].

06(PCTS_GTI_DEVICE?C:UNTESTED)

If either the behavior associated with {_POSIX_JOB_CONTROL} or *tcsetpgrp*() is supported, and the implementation supports an invalid value for *pgrp_id*:

When *pgrp_id* is a value not supported by the implementation, then a call to *tcsetpgrp*(*fildes, pgrp_id*) returns a value of (*int*)-1, sets *errno* to [EINVAL], and does not change the process group ID of the controlling terminal associated with *fildes*.

07(C) If both the behavior associated with (_POSIX_JOB_CONTROL} and *tcsetpgrp*() are not supported:
A call to *tcsetpgrp*() is unsuccessful, returns a value of (*int*)-1, and sets *errno* to [ENOSYS].

08(PCTS_GTI_DEVICE?C:UNTESTED)

If either the behavior associated with {_POSIX_JOB_CONTROL} or *tcsetpgrp*() is supported:
When the calling process does not have a controlling terminal, then a call to *tcsetpgrp*(*fildes, pgrp_id*) returns a value of (*int*)-1, sets *errno* to [ENOTTY], and does not change the process group ID of the controlling terminal associated with *fildes*.

09(C) If either the behavior associated with {_POSIX_JOB_CONTROL} or *tcsetpgrp*() is supported:
When *fildes* is not associated with the controlling terminal, then a call to *tcsetpgrp*(*fildes, pgrp_id*) returns a value of (*int*)-1 and sets *errno* to [ENOTTY].
Testing Requirements:
Test for a file that is not a terminal and, if the implementation provides device types that support the general terminal interface, for a terminal that is not the controlling terminal of the process.

10(PCTS_GTI_DEVICE?C:UNTESTED)

If either the behavior associated with {_POSIX_JOB_CONTROL} or *tcsetpgrp*() is supported:
When the controlling terminal is no longer associated with the session of the calling process, then a call to *tcsetpgrp*(*fildes, pgrp_id*) returns a value of (*int*)-1, sets *errno* to [ENOTTY], and does not change the process group ID of the controlling terminal associated with *fildes.*

11(PCTS_GTI_DEVICE?C:UNTESTED)

If either the behavior associated with {_POSIX_JOB_CONTROL} or *tcsetpgrp*() is supported:
When the value of *pgrp_id* does not match the process group ID of a process in the same session as the calling process, then a call to *tcsetpgrp*(*fildes, pgrp_id*) returns a value of (*int*)-1, sets *errno* to [EPERM], and does not change the process group ID of the controlling terminal associated with *fildes.*

8. Language-Specific Services for the C Programming Language

8.1 Referenced C Language Routines

NOTE — Implementations claiming conformance to Common-Usage C shall adhere to the requirements stated in 1.4.19.

D01(C) If the implementation provides Common-Usage C support and does not support a C Standard {2} function or deviates from the C Standard {2}:
These differences are documented in the appropriate section of the PCD.1.

GA63 Implementations shall comply with the requirements outlined for the C Standard {2}.

8.1.1 *assert*() (8.1)

8.1.1.1 Synopsis (8.1)

01(A) When the header `<assert.h>` is included, then *assert*() is defined as a macro that expands to an expression with the result type *void*. (See GA36 in 2.7.3.)

8.1.1.2 Description (8.1)

02(B) If the implementation provides C Standard {2} support:

Element *assert*() complies with the requirements of the C Standard {2}. (See GA63 in 8.)

Otherwise:

Element *assert*() complies with the requirements of' the C Standard {2} unless amended by the PCD.1. (See GA63 in 8..)

See Reason 2 in Section 5. of POSIX.3 {4}.

8.1.1.3 Returns (8.1)

There are no assertions specific to this subclause.

8.1.1.4 Errors (8.1)

There are no assertions specific to this subclause.

8.1.2 Character Handling Functions (8.1)

8.1.2.1 Synopsis (8.1)

01(A) For *funct*() of *isalnum*(), *isalpha*(), *iscntrl*(), *isdigit*(), *isgraph*(), *islower*(), *isprint*(), *ispunct*(), *isspace*(), *isupper*(), *isxdigit*(), *tolower*(), and *toupper*():

If the implementation provides C Standard {2} support:

When the header `<ctype.h>` is included, then the function prototype `int funct(int)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<ctype.h>` is included, then the function *funct*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) For *funct*() of *isalnum*(), *isalpha*(), *iscntrl*(), *isdigit*(), *isgraph*(), *islower*(), *isprint*(), *ispunct*(), *isspace*(), *isupper*(), *isxdigit*(), *tolower*(), and *toupper*():

If *funct*() is defined as a macro when the header `<ctype.h>` is included:

When the macro *funct*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If the *isalnum*(), *isalpha*(), *isdigit*(), *isgraph*(), *islower*(), *isprint*(), *ispunct*(), *isspace*(), *isupper*(), *isxdigit*(), *tolower*(), or *toupper*() function is defined as a macro in the header `<ctype.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and yields a result value that can be used in complex expressions without requiring extra parentheses. (See GA01 in 1.3.4.)

8.1.2.2 Description (8.1)

04(B) For *funct*() of *isalnum*(), *isalpha*(), *iscntrl*(), *isdigit*(), *isgraph*(), *islower*(), *isprint*(), *ispunct*(), *isspace*(), *isupper*(), *isxdigit*(), *tolower*(), and *toupper*():

If the implementation provides C Standard {2} support:

Element *funct*() complies with the requirements of the C Standard {2}. (See GA63 in 8..)

Otherwise:

Element *funct*() complies with the requirements of the C Standard {2} unless amended by the PCD.1. (See GA63 in 8.)

See Reason 2 in Section 5. of POSIX.3 {4}.

8.1.2.3 Returns (8.1)

There are no assertions specific to this subclause.

8.1.2.4 Errors (8.1)

There are no assertions specific to this subclause.

8.1.3 *setlocale*() (8.1)

8.1.3.1 Synopsis (8.1)

01(A) If the implementation provides C Standard {2} support:
When the header `<locale.h>` is included, then the function prototype `char * setlocale(int, const char *)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<locale.h>` is included, then the function *setlocale*() is declared with the result type *char* *. (See GA36 in 2.7.3.)

02(C) If *setlocale*() is defined as a macro when the header `<locale.h>` is included:
When the macro *setlocale*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *char* *. (See GA37 in 2.7.3.)

03(C) If *setlocale*() is defined as a macro in the header `<locale.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

8.1.3.2 Description (8.1.2.2)

D01(C) If the implementation supports values for *locale* other than "C":
The other values supported for *locale* are documented in 8.1.2.2 of the PCD.1.

04(A) When *locale* is a pointer to the string "C", then a call to *setlocale*(*category*, *locale*) sets the locale for the specified category to the minimal environment for C-language translation.

NOTE — The minimal environment for C-language translation is specified in Section 4.4 of the C Standard {2}.

05(A) When *locale* is a pointer to the string "POSIX", then a call to *setlocale*(*category*, *locale*) sets the locale for the specified category to an environment that includes the minimal environment, for C-language translation.

06(A) When no call has been made to *setlocale*() by the application after the most recent successful replacement of the process image, then the "C" locale is the locale associated with the process.

D02(C) If the string returned from *setlocale*(), when *locale* is a **NULL** pointer, is documented:
The details are contained in 8.1.2.2 of the PCD.1.

07(A) When *locale* is a pointer to a NULL string, and when the environment variable **LC_ALL** specifies a valid locale, then *setlocale*(*category*, *locale*) sets the locale for the requested category (or all categories in the case of category **LC_ALL**) to the value of the environment variable **LC_ALL**.

08(A) When *locale* is a pointer to a NULL string, and when the environment variable **LC_ALL** is not a NULL string and does not specify a valid locale, then *setlocale*(*category*, *locale*) returns NULL and does not change the locale setting.

D03(C) If additional categories are supported for the *setlocale*() function:
These categories are documented in 8.1.2.2 of the PCD.1.

D04(A) When the **LC_ALL** environment variable is not set or is set to the empty string, when the environment variable corresponding to the specified category is not set or is set to the null string, and when the **LANG** environment variable is not set or is set to the empty string, whether *setlocale*() sets the specified category of the locale of the process to a system wide default value, to "C", or to "POSIX" is stated in 8.1.2.2 of the PCD.1.

09(A) When *category* is any of **LC_CTYPE**, **LC_COLLATE**, **LC_TIME**, **LC_NUMERIC**, or **LC_MONETARY**, when *locale* is a pointer to a NULL string, when the environment variable **LC_ALL** is either not set or is set to a NULL string, and when the environment variable corresponding to *category* is set to a valid locale, then *setlocale*(*category*, *locale*) sets the specified category to the value in the corresponding environment variable.

10(A) When *category* is any of **LC_CTYPE**, **LC_COLLATE**, **LC_TIME**, **LC_NUMERIC**, or **LC_MONETARY**, when *locale* is a pointer to a NULL string, when the environment variable **LC_ALL** is either not set or is set to a NULL string, and when the environment variable corresponding to *category* is not a NULL string and is not a valid locale, then *setlocale*(*category*, *locale*) returns NULL and does not change the locale setting.

11(A) When *category* is any of **LC_CTYPE**, **LC_COLLATE**, **LC_TIME**, **LC_NUMERIC**, or **LC_MONETARY**, when *locale* is a pointer to a NULL string, when the environment variable **LC_ALL** and the environment variable corresponding to *category* are each either not set or set to a NULL string, and when the environment variable **LANG** is set to a valid locale, then *setlocale*(*category*, *locale*) sets the specified category to the value in the **LANG** environment variable.

12(A) When *category* is any of **LC_CTYPE**, **LC_COLLATE**, **LC_TIME**, **LC_NUMERIC**, or **LC_MONETARY**, when *locale* is a pointer to a NULL string, when the environment variable **LC_ALL** is either not set or is set to NULL, when the corresponding environment variable is either not set or is set to a NULL string, and when the environment variable **LANG** is not a NULL string and does not specify a valid locale, then *setlocale*(*category*, *locale*) returns NULL and does not change the locale setting.

13(A) When *category* is **LC_ALL**, when locale is a pointer to a NULL string, and when the environment variable **LC_ALL**. is either not set or is set to a NULL string, then for each of the environment variables **LC_CTYPE**, **LC_COLLATE**, **LC_TIME**, **LC_NUMERIC**, and **LC_MONETARY** that are not NULL strings and that specify a valid locale, *setlocale*(*category*, *locale*) sets the locale for that *category* to the value of the corresponding environment variable.

14(A) When *category* is **LC_ALL**, when *locale* is a pointer to a NULL string, when the environment variable **LC_ALL** is either not set or is set to a NULL string, and when the environment variable **LANG** is set to a valid locale, then, for each of the environment variables **LC_CTYPE**, **LC_COLLATE**, **LC_TIME**, **LC_NUMERIC**, and **LC_MONETARY** that is either not set or set to a NULL string, *setlocale*(*category*, *locale*) sets the *locale* for that category to the value of the environment variable **LANG**.

15(A) When *category* is **LC_ALL**, when *locale* is a pointer to a NULL string, when the environment variable **LC_ALL** is either not set or is set to NULL, and when any of the environment variables **LC_CTYPE**, **LC_COLLATE**, **LC_TIME**, **LC_NUMERIC**, and **LC_MONETARY** is not a NULL string and does not specify a valid locale, then *setlocale*(*category*, *locale*) returns NULL and does not change the locale setting.

16(A) When *category* is **LC_ALL**, when *locale* is a pointer to a NULL string, when the environment variable **LC_ALL** is either not set or is set to a NULL, when any of the environment variables **LC_CTYPE**, **LC_COLLATE**, **LC_TIME**, **LC_NUMERIC**, and **LC_MONETARY** is either not set or is set to a NULL string, and when the environment variable **LANG** is not a NULL string and does not specify a valid locale, then *setlocale*(*category*, *locale*) returns NULL and does not change the locale setting.

8.1.3.3 Returns (8.1)

There are no assertions specific to this subclause.

8.1.3.4 Errors (8.1)

There are no assertions specific to this subclause.

8.1.4 Mathematics Functions (8.1)

8.1.4.1 Synopsis (8.1)

01(A) For *funct*() of *acos*(), *asin*(), *atan*(), *cos*(), *sin*(), *tan*(), *cosh*(), *sinh*(), *tanh*(), *exp*(), *log*(), *log10*(), *sqrt*(), *ceil*(), *fabs*(), and *floor*():

If the implementation provides C Standard {2} support:

When the header `<math.h>` is included, then the function prototype `double funct(double)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<math.h>` is included, then the function *funct*() is declared with the result type *double*. (See GA36 in 2.7.3.)

For *funct*() of *atan2*(), *pow*(), and *fmod*():

If the implementation provides C Standard {2} support:

When the header `<math.h>` is included, then the function prototype `double funct(double, double)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<math.h>` is included, then the function *funct*() is declared with the result type *double*. (See GA36 in 2.7.3.)

For *frexp*():

If the implementation provides C Standard {2} support:

When the header `<math.h>` is included, then the function prototype `double frexp(double, int *)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<math.h>` is included, then the function *frexp*() is declared with the result type *double*. (See GA36 in 2.7.3.)

For *ldexp*():

If the implementation provides C Standard {2} support:

When the header `<math.h>` is included, then the function prototype `double ldexp(double, int)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<math.h>` is included, then the function *ldexp*() is declared with the result type *double*. (See GA36 in 2.7.3.)

For *modf*():

If the implementation provides C Standard {2} support:

When the header `<math.h>` is included, then the function prototype `double modf(double, double *)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<math.h>` is included, then the function *modf*() is declared with the result type *double*. (See GA36 in 2.7.3.)

02(C) For *funct*() of *acos*(), *asin*(), *atan*(), *cos*(), *sin*(), *tan*(), *cosh*(), *sinh*(), *tanh*(), *exp*(), *log*(), *log10*(), *sqrt*(), *ceil*(), *fabs*(), *floor*(), *atan2*(), *pow*(), *fmod*(), *frexp*(), *ldexp*(), and *modf*():

If *funct*() is defined as a macro when the header `<math.h>` is included:

When the macro *funct*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type double. (See GA37 in 2.7.3.)

03(C) If the *acos*(), *asin*(), *atan*(), *cos*(), *sin*(), *tan*(), *cosh*(), *sinh*(), *tanh*(), *exp*(), *log*(), *log10*(), *sqrt*(), *ceil*(), *fabs*(), *floor*(), *atan2*(), *pow*(), *fmod*(), *frexp*(), *ldexp*(), or *modf*() function is defined as a macro in the header `<math.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary and yields a result value that can be used in complex expressions without requiring extra parentheses. (See GA01 in 1.3.4.)

8.1.4.2 Description (8.1)

04(B) For *funct*() of *acos*(), *asin*(), *atan*(), *cos*(), *sin*(), *tan*(), *cosh*(), *sinh*(), *tanh*(), *exp*(), *log*(), *log10*(), *sqrt*(), *ceil*(), *fabs*(), *floor*(), *atan2*(), *pow*(), *fmod*(), *frexp*(), *ldexp*(), and *modf*():

If the implementation provides C Standard {2} support:

Element *funct*() complies with the requirements of the C Standard {2}. (See GA63 in 8.)

Otherwise:

Element *funct*() complies with the requirements of the C Standard {2} unless amended by the PCD.1. (See GA63 in 8..)

See Reason 2 in Section 5. of POSIX.3 {4}.

8.1.4.3 Returns (8.1)

There are no assertions specific to this subclause.

8.1.4.4 Errors (8.1)

There are no assertions specific to this subclause.

8.1.5 *setjmp*() (8.1)

8.1.5.1 Synopsis (8.1)

01(C) If *setjmp*() is not defined as a macro:

When the header `<setjmp.h>` is included, then *setjmp*() is declared as an identifier with external linkage and result type *int*. (See GA36 in 2.7.3.)

02(C) If *setjmp*() is defined as a macro in the header `<setjmp.h>`:

When the header `<setjmp.h>` is included and the macro *setjmp*() is invoked with the correct argument types, then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *setjmp*() is defined as a macro in the header `<setjmp.h>`:

It protects its return value with extra parentheses when necessary. (See GA01 in 1.3.4.)

8.1.5.2 Description (8.1)

04(B) If the implementation provides C Standard {2} support:

Element *setjmp*() complies with the requirements of the C Standard {2}. (See GA63 in 8..)

Otherwise:

Element *setjmp*() complies with the requirements of the C Standard {2} unless amended by the PCD.1. (See GA63 in 8.)

See Reason 2 in Section 5. of POSIX.3 {4}.

8.1.5.3 Returns (8.1)

There are no assertions specific to this subclause.

8.1.5.4 Errors (8.1)

There are no assertions specific to this subclause.

8.1.6 *longjmp*() (8.1)

8.1.6.1 Synopsis (8.1)

01(C) If the implementation provides C Standard {2} support:
When the header `<setjmp.h>` is included, then the function prototype `void longjmp(jmp_buf, int)` is declared. (See GA36 in 2.7.3.)

D01(C) If the implementation provides Common-Usage C support:
The result type for function *longjmp*() is contained in 8.1 of the PCD.1. (See DGA01 in 1.3.3.3.)

02(D) If the implementation provides C Standard {2} support and *longjmp*() is defined as a macro when the header `<setjmp.h>` is included:
When the macro *longjmp*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *void*. (See GA37 in 2.7.3.)

See Reason 2 in Section 5. of POSIX.3 {4}.

03(C) If *longjmp*() is defined as a macro in the header `<setjmp.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary. (See GA01 in 1.3.3.)

8.1.6.2 Description (8.1)

04(B) If the implementation provides C Standard {2} support:
Element *longjmp*() complies with the requirements of the C Standard {2}. (See GA63 in 8.)

Otherwise:
Element *longjmp*() complies with the requirements of the C Standard {2} unless amended by the PCD.1. (See GA63 in 8.)

See Reason 2 in Section 5. of POSIX.3 {4}.

8.1.6.3 Returns (8.1)

There are no assertions specific to this subclause.

8.1.6.4 Errors (8.1)

There are no assertions specific to this subclause.

8.1.7 *clearerr*() (8.1)

8.1.7.1 Synopsis (8.1)

01(C) If the implementation provides C Standard {2} support:
When the header `<stdio.h>` is included, then the function prototype `void clearerr(FILE *)` is declared. (See GA36 in 2.7.3.)

D01(C) If the implementation provides Common-Usage C support:
The result type for function *clearerr*() is contained in 8.1 of the PCD.1. (See DGA01 in 1.3.3.3.)

02(D) If the implementation provides C Standard {2} support and *clearerr*() is defined as a macro when the header `<stdio.h>` is included:

When the macro *clearerr*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *void*. (See GA37 in 2.7.3.)

See Reason 2 in Section 5. of POSIX.3 {4}.

03(C) If *clearerr*() is defined as a macro in the header `<stdio.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary. (See GA01 in 1.3.3.)

8.1.7.2 Description (8.1)

04(B) If the implementation provides C Standard {2} support:

Element *clearerr*() complies with the requirements of the C Standard {2}. (See GA63 in 8..)

Otherwise:

Element *clearerr*() complies with the requirements of the C Standard {2} unless amended by the PCD.1. (See GA63 in 8..)

See Reason 2 in Section 5. of POSIX.3 {4}.

8.1.7.3 Returns (8.1)

There are no assertions specific to this subclause.

8.1.7.4 Errors (8.1)

There are no assertions specific to this subclause.

8.1.8 *fclose*() (8.1)

8.1.8.1 Synopsis (8.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<stdio.h>` is included, then the function prototype `int fclose(FILE *)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<stdio.h>` is included, then the function *fclose*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *fclose*() is defined as a macro when the header `<stdio.h>` is included:

When the macro *fclose*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *fclose*() is defined as a macro in the header `<stdio.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

8.1.8.2 Description (8.2.3.2)

04(A) The *fclose*() function deallocates the file descriptor underlying the stream indicated by its argument (i.e., this file descriptor will be made available for subsequent *open*()s by the process), and a value of zero is returned.

05(A) When a call to *fclose*() on the stream is performed, then all outstanding record locks owned by the process on the file associated with the stream are removed.

R01 When a call to *fclose*() is interrupted by a signal that is to be caught, then the call to *fclose*() returns a value of (*int*)−1 and sets *errno* to [EINTR]. (See Assertion 15 in 8.1.8.4.)

06(B) When all file descriptors associated with a pipe have been closed by *fclose*(), then data remaining in the pipe is discarded.

See Reason 3 in Section 5. of POSIX.3 {4}.

07(A) When all file descriptors associated with a FIFO special file have been closed by *fclose*(), then data remaining in the FIFO is discarded.

08(B) When all file descriptors associated with an open file description have been closed by *fclose*(), then the open file description is freed.

See Reason 1 in Section 5. of POSIX.3 {4}.

09(B) When all file descriptors associated with a file have been closed by *fclose*(), and when the link count of the file is zero, then the space occupied by the file is freed.

See Reason 1 in Section 5. of POSIX.3 {4}.

10(B) When all file descriptors associated with a file have been closed by *fclose*(), and when the link count of the file is zero, then the file is no longer accessible.

See Reason 3 in Section 5. of POSIX.3 {4}.

11(A) When a file is closed via *fclose*() by the last process that had it open, all time-related fields still marked for update are updated.

12(A) When the stream is writable and buffered data is caused to be written, then a call to *fclose*(*stream*) marks for update the *st_ctime* and *st_mtime* fields of the underlying file.

8.1.8.3 Returns (8.2.3.11)

R02 When a call to *fclose*() completes unsuccessfully, then a value of EOF is returned and sets *errno* to indicate the error. (See Assertions 13-19 in 8.1.8.4.)

8.1.8.4 Errors (8.2.3.11)

3(C) If the implementation provides buffered streams:

When the stream references a pipe or FIFO, and when the underlying file descriptor has the O_NONBLOCK flag set and contains insufficient capacity to accept the buffered data, then a call to *fclose*() returns a value of EOF and sets *errno* to [EAGAIN].

Testing Requirements:

Test for both a pipe and a FIFO.

14(A) When the stream pointer argument addresses an underlying file descriptor that is closed, then a call to *fclose*() returns a value of EOF and sets *errno* to [EBADF].

15(C) If the implementation provides buffered streams:

When *fclose*() is terminated due to receipt of a signal, and when no data was transferred, then the call to *fclose*() returns a value of EOF and sets *errno* to [EINTR].

16(D) If the implementation provides buffered streams:

When attempting to write to a file, and when flushing the output buffer would cause the size of the file to exceed an implementation-defined maximum file size, then a call to *fclose*() returns a value of EOF and sets *errno* to [EFBIG].

See Reason 3 in Section 5. of POSIX.3 {4}.

17(D) If the implementation provides buffered streams:

When attempting to write to a device, and when flushing the output buffer would exceed the amount of space available for data on that device, then a call to *fclose*() returns a value of EOF and sets *errno* to [ENOSPC].

See Reason 1 in Section 5. of POSIX.3 {4}.

18(C) If the implementation provides buffered streams:

When attempting to write to a pipe or a FIFO that is not open for reading by any process, then a call to *fclose*() returns a value of EOF, sets *errno* to [EPIPE], and sends a SIGPIPE signal to the calling process.

Testing Requirements:

Test for both a pipe and a FIFO.

19(PCTS_GTI_DEVICE?C:UNTESTED)

If both the behavior associated with {_POSIX_JOB CONTROL} is supported and the implementation provides buffered streams:

When the process is neither ignoring nor blocking the SIGTTOU signal and is a member of an orphaned background process group, and when TOSTOP is set, then a call to *fclose*() returns a value of EOF and sets *errno* to [EIO].

8.1.9 *feof*() (8.1)

8.1.9.1 Synopsis (8.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<stdio.h>` is included, then the function prototype `int feof(FILE *)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<stdio.h>` is included, then the function *feof*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *feof*() is defined as a macro when the header `<stdio.h>` is included:

When the macro *feof*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *feof*() is defined as a macro in the header `<stdio.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

8.1.9.2 Description (8.1)

04(A) A call to *feof*(*stream*) returns nonzero when the end of file indicator associated with stream is set.

05(A) A call to *feof*(*stream*) returns zero when the end of file indicator associated with stream is clear.

8.1.9.3 Returns (8.1)

There are no assertions specific to this subclause.

8.1.9.4 Errors (8.1)

There are no assertions specific to this subclause.

8.1.10 *ferror*() (8.1)

8.1.10.1 Synopsis (8.1)

01(A) If the implementation provides C Standard {2} support:
When the header `<stdio.h>` is included, then the function prototype `int ferror(FILE *)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<stdio.h>` is included, then the function *ferror*() is either declared with result type int or not declared in the header. (see GA36 in 2.7.3.)

02(C) If *ferror*() is defined as a macro when the header `<stdio.h>` is included:
When the macro *ferror*() is invoked with the correct, argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (see GA37 in 2.7.3.)

03(C) If *ferror*() is defined as a macro in the header `<stdio.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

8.1.10.2 Description (8.1)

04(A) A call to *ferror*(*stream*) returns nonzero when the error indicator associated with *stream* is set.

05(A) A call to *ferror*(*stream*) returns zero when the error indicator associated with *stream* is clear.

8.1.10.3 Returns (8.1)

There are no assertions specific to this subclause.

8.1.10.4 Errors (8.1)

There are no assertions specific to this subclause.

8.1.11 *fflush*() (8.1)

8.1.11.1 Synopsis (8.1)

01(A) If the implementation provides C Standard {2} support:
When the header `<stdio.h>` is included, then the function prototype `int fflush(FILE *)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<stdio.h>` is included, then the function *fflush*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *fflush*() is defined as a macro when the header `<stdio.h>` is included:
When the macro *fflush*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *fflush*() is defined as a macro in the header `<stdio.h>`:
It evaluates its arguments only once, fully protracted by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

8.1.11.2 Description (8.2.3.4)

04(A) When the stream is writable and buffered data is caused to be written, then a call to *fflush*() marks for update the *st_ctime* and *st_mtime* fields of the underlying file.

8.1.11.3 Returns (8.2.3.11)

R01 When a call to *fflush*() completes unsuccessfully, then a value of EOF is returned and sets *errno* to indicate the error. (See Assertions 5-11 in 8.1.11.4.)

8.1.11.4 Errors (8.2.3.11)

05(C) If the implementation provides buffered streams:
When the underlying file descriptor references a pipe or a FIFO that has the O_NONBLOCK flag set and contains insufficient capacity to accept the buffered data, then a call to *fflush*() returns a value of EOF and sets *errno* to [EAGAIN].
Testing Requirements:
Test for both a pipe and a FIFO.

06(A) When the stream pointer argument addresses a file descriptor that is closed, then a call to *fflush*() returns a value of EOF and sets *errno* to [EBADF].

07(C) If the implementation provides buffered streams:
When *fflush*() is terminated due to receipt of a signal, and when no data was transferred, then the call to *fflush*() returns a value of EOF and sets *errno* to [EINTR].

08(D) If the implementation provides buffered streams:
When attempting to write to a file that exceeds an implementation-defined maximum file size, then a call to *fflush*() returns a value of EOF and sets *errno* to [EFBIG].

See Reason 3 in Section 5. of POSIX.3 {4}.

09(D) If the implementation provides buffered streams:
When attempting to write to a device that has no more space for en a call to *fflush*() returns a value of EOF and sets *errno* to [ENOSPC].

See Reason 1 in Section 5. of POSIX.3 {4}.

10(C) If the implementation provides buffered streams:
When attempting to write to a pipe or a FIFO t hat is not open for reading by any process, then a call to *fflush*() returns a value of EOF, sets *errno* to [EPIPE], and sends a SIGPIPE signal to the calling process.
Testing Requirements:
Test for both a pipe and a FIFO.

11(PCTS_GTL_DEVICE?C:UNTESTED)

If both the behavior associated with {_POSIX_JOB_CONTROL} is supported and the implementation provides buffered streams:
When the process is neither ignoring nor blocking the SIGTTOU signal and is a member of an orphaned background process group, and when TOSTOP is set, then a call to *fflush*() returns a v alue of EOF and sets *errno* to [EIO].

8.1.12 *fgetc*() (8.1)

8.1.12.1 Synopsis (8.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<stdio.h>` is included, then the function prototype `int fgetc(FILE *)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<stdio.h>` is included, then the function *fgetc*() is either declared with result type *int* not declared in the header. (See GA36 in 2.7.3.)

02(C) If *fgetc*() is defined as a macro when the header `<stdio.h>` is included:

When the macro *fgetc*() is invoked with the correct argument types (or compatible argument types in the case that C Standard 2} support is provided), then it expands to an expression with the result type *int.* (See GA37 in 2.7.3.)

03(C) If *fgetc*() is defined as a macro in the header `<stdio.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (see GA01 in 1.3.4.)

8.1.12.2 Description (8.2.3.5)

04(A) When data was not supplied by a prior call to *ungetc*(), and when there has been no previous buffered read operation on the stream, then a call to *fgetc*() marks for update the *st_atime* field of the underlying file.

8.1.12.3 Returns (8.2.3.11)

R01 When a call to *fgetc*() completes unsuccessfully, then a value of EOF is returned and sets *errno* to indicate the error. (See Assertions 5-9 in 8.1.12.4.)

8.1.12.4 Errors (8.2.3.11)

05(A) When the underlying file descriptor references a pipe or a FIFO that has the O_NONBLOCK flag set and contains insufficient data to process a read request, then a call to *fgetc*() returns a value of EOF and sets *errno* to [EAGAIN].

Testing Requirements:

Test for both a pipe and a FIFO.

06(A) When the stream pointer argument addresses a file descriptor that is not open for reading, then a call to *fgetc*() returns a value of EOF and sets *errno* to [EBADF].

07(A) When *fgetc*() is terminated due to receipt of a signal, and when no data was transferred, then the call to *fgetc*() returns a value of EOF and sets *errno* to [EINTR].

08(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When the process is ignoring or blocking the SIGTTIN signal and is a member of a background process group, and when the *stream* argument references the controlling terminal, then a call to *fgetc*(*stream*) returns a value of EO and sets *errno* to [EIO].

09(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When the process is a member of an orphaned background process group, and when the *stream*argument references the controlling terminal, then a call to *fgetc*(*stream*) returns a value of EOF and sets *errno* to [EIO].

8.1.13 *fgets*() (8.1)

8.1.13.1 Synopsis (8.1)

01(A) If the implementation provides C Standard {2} support:

When the header <stdio.h> is included, then the function prototype
`char * fgets(char *,int, FILE *)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header <stdio.h> is included, then the function *fgets*() is declared with the result type *char* *. (See GA36 in 2.7.3.)

02(C) If *fgets*() is defined as a macro when the header <stdio.h> is included:

When the macro *fgets*() is invoked with the correct argument types (or compatible argument types in the case that C Standard 2} support is provided), then it expands to an expression with the result type *char**. (See GA37 in 2.7.3.)

03(C) If *fgets*() is defined as a macro in the header <stdio.h>:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

8.1.13.2 Description (8.2.3.5)

04(A) On a call to *fgets*() when data was not supplied by a prior call to *ungetc*(), and when there has been no previous buffered read operation on the stream, then the *st_atime* field of the underlying file is marked for update.

8.1.13.3 Returns (8.2.3.11)

R01 When a call to *fgets*() completes unsuccessfully, then a **NULL** pointer is returned and sets *errno* to indicate the error. (See Assertions 5-9 in 8.1.13.4.)

8.1.13.4 Errors (8.2.3.11)

05(A) When the underlying file descriptor references a pipe or a FIFO that has the O_NONBLOCK flag set and contains insufficient data to process a read request, then a call to *fgets*() returns a **NULL** pointer and sets *errno* to [EAGAIN].

Testing Requirements:

Test for both a pipe and a FIFO.

06(A) When the stream pointer argument addresses a file descriptor that is not open for reading, then a call to *fgets*() returns a **NULL** pointer and sets *errno* to [EBADF].

07(A) When *fgets*() is terminated due to receipt of a signal, and when no data was transferred, then the call to *fgets*() returns a **NULL** pointer and sets *errno* to [EINTR].

08(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When the process is ignoring or blocking the SIGTTIN signal and is a member of a background process group, and when the *stream* argument references the controlling terminal, then a call to *fgets*(*stream*) returns a **NULL** pointer and sets *errno* to [EIO].

09(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When the process is a member of an orphaned background process group, and when the *stream* argument references the controlling terminal, then a call to *fgets*(*stream*) returns a **NULL** pointer and sets *errno* to [EIO].

8.1.14 *fopen*() (8.1)

8.1.14.1 Synopsis (8.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<stdio.h>` is included, then the function prototype `FILE * fopen(const char *, const char *)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<stdio.h>` is included, then the function *fopen*() is declared with the result type *FILE* *. (See GA36 in 2.7.3.)

02(C) If *fopen*() is defined as a macro when the header `<stdio.h>` is included:

When the macro *fopen*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *FILE* *. (See GA37 in 2.7.3.)

03(C) If *fopen*() is defined as a macro in the header `<stdio.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

8.1.14.2 Description (8.2.3.1)

04(A) When the first filename component of the *filename* argument is "", and the pathname does not begin with a slash, then *fopen*() resolves the pathname by locating the second filename component when specified) in the current working directory. (See GA14 in 2.3.6.)

05(A) When the *filename* argument points to the string "`/`", then *fopen*() resolves the pathname to the root directory of the process. (See GA15 in 2.3.6.)

06(A) When the *filename* argument points to the string "`///`", then *fopen*() resolves the pathname to the root directory of the process. (See GA16 in 2.3.6.)

07(A) When the *filename* argument points to a string beginning with a single slash or beginning with three or more slashes, then *fopen*() resolves the pathname by locating the first filename component of the pathname in the root directory of the process. (See GA17 in 2.3.6.)

08(A) When the first filename component of the *filename* argument is "..", the pathname does not begin with a slash, and the current working directory is not the root directory of the process, then *fopen*() resolves the pathname by locating the second filename component (when specified) in the parent directory of the current working directory. (See GA18 in 2.3.6.)

09(A) When the *filename* argument points to the string "F1/" and `F1` is a directory, then *fopen*() resolves the pathname to `F1`. (See GA19 in 2.3.6.)

10(A) When the argument *filename* points to the string "F1//" and `F1` is a directory, then *fopen*() resolves the pathname to `F1`, which is in the current working directory. (See GA20 in 2.3.6.)

11(A) When the *filename* argument points to the string "F1/F2", then *fopen*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA21 in 2.3.6.)

12(A) When the pathname argument points to the string "F1/./F2", then *fopen*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA22 in 2.3.6.)

13(A) When the *filename* argument points to the string "F1/../F1/F2", then *fopen*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA23 in 2.3.6.)

14(A) When the *filename* argument points to the string "F1//F2", then *fopen*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA24 in 2.3.6.)

15(C) If {_POSIX_NO_TRUNC} is not supported in the corresponding directory:

When the *filename* component is a string of more than {NAME_MAX} bytes in a directory for which {_POSIX_NO_TRUNC} is not supported, then *fopen*() resolves the pathname component by truncating it to {NAME_MAX} bytes. (See GA25 in 2.3.6.)

16(A) On a call to *fopen*(*filename*, *type*), the pathname *filename* supports *filenames* containing any of the following characters:

ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghij klmnopqrstuvwxyz
0123456789._-

(See GA02 in 2.2.2.32.)

17(A) On a call to *fopen*(*filename*, *type*), the pathname *filename* retains the unique identity of its upper- and lowercase letters. (See GA03 in 2.2.2.32.)

18(A) A call to *fopen*() allocates a file descriptor as a call to *open*() does.

19(A) When a call to *fopen*() creates a file, then the file permission bits are set to allow both reading and writing for owner, for group and for other users except for those bits set in the file mode creation mask of the process. No execute (search) permission bits are set.

8.1.14.3 Returns (8.2.3.11)

R01 When a call to *fopen*() completes unsuccessfully, then a **NULL** pointer is returned and sets *errno* to indicate the error. (See Assertions 20-24, 26, and 29-34 in 8.1.14.4.)

8.1.14.4 Errors (8.2.3.11)

20(A) When search permission is denied on a component of the *filename* prefix, then a call to *fopen*(*filename*, *type*) returns a **NULL** pointer and sets *errno* to [EACCES].

21(A) When the file exists and the requested read and/or write permission is denied, then a call to *fopen*(*filename*, *type*) returns a **NULL** pointer and sets *errno* to [EACCES].

Testing Requirements:

Test for instances, when *type* is “w”, “w+”, “wb”, “wb+”, or “w+b”, that the file is not truncated.”

22(A) When the file does not exist, and when write permission is denied for the parent directory of the file to be created, then a call to *fopen*(*filename*, *type*) returns a **NULL** pointer, sets *errno* to [EACCES], and the file is not created.

23(A) When *fopen*() is interrupted by a signal, then the call to *fopen*(*filename*, *type*) returns a **NULL** pointer, sets *errno* to [EINTR], and does not mark for update *st_ctime* and *st_mtime* fields of the file and parent directory and the *st_atime* field of the file.

24(A) When the named file is a directory, and when *type* is “w”, “a”, “r+”, “w+”, “a+”, “wb”, “ab”, “r+b”, “rb+”, “w+b”, “wb+”, “a+b”, or “ab+”, then a call to *fopen*(*filename*, *type*) returns a **NULL** pointer, sets *errno* to [EISDIR], and does not mark for update *st_ctime*, *st_mtime*, and *st_atime* fields of the file.

25(A) If {OPEN_MAX}≤{PCTS_OPEN_MAX}:

When {OPEN_MAX} file descriptors have been opened, then a subsequent call to *fopen*(*filename*, *type*, *mode*) returns a **NULL** pointer ans sets *errno* to [EMFILE].

Testing Requirements:

Test for instances

— When the file does not exist and *type* is “w”, “a”, “w+”, “a+”, “wb”, “ab”, “wb+”, “ab+”, “w+b”, or “a+b”, that the file is not created, and

— When the file exists and *type* is “w”, “w+”, “wb”, “wb+”, or “w+b”, that the file is not truncated

Otherwise:

{PCTS_OPEN_MAX} files can be opened.

26({NAME_MAX} ≤ {PCTS_NAME_MAX}?C:UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the underlying file:

When the length of a pathname component of *filename* is longer than the maximum number of bytes in a filename {NAME_MAX}, then a call to *fopen*(*filename*, *type*) returns a value of (*int*)-1 and sets *errno* to [ENAMETOOLONG].

27({NAME_MAX}>{PCTS_NAME_MAX}?C :UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the underlying file:

When the length of a pathname component of *filename* equals {PCTS_NAME_MAX}, then a call to *fopen*(*filename*, *type*) succeeds.

28(A) If {PATH_MAX} ≤ {PCTS_PATH_MAX}:

When the length of the *filename* argument exceeds the maximum number of bytes in a pathname {PATH_MAX}, then a call to *fopen*(*filename*, *type*) returns a **NULL** pointer and sets *errno* to [ENAMETOOLONG].

Otherwise:

A filename of length {PCTS_PATH_MAX} can be opened.

29(B) When too many files are currently open in the system, then a call to *fopen*(*filename*, *type*) returns a **NULL** pointer and sets *errno* to [ENFILE].

Testing Requirements:

Test for instances

— When the file does not exist and *type* is “w”, “a”, “w+”, “a+”, “wb”, “ab”, “wb+”, “ab+”, “w+b”, or “a+b”, that the file is not created, and

— When the file exists and *type* is “w”, “w+”, “wb”, “wb+”, or “w+b”, that the file is not truncated

See Reason 1 in Section 5. of POSIX.3 {4}.

30(A) When *type* is “r”, “r+”, “rb”, “r+b”, or “rb+”, and when the named file does not exist, then a call to *fopen*(*filename*, *type*) returns a **NULL** pointer and sets *errno* to [ENOENT].

31(A) When a component of the path prefix of *filename* does not exist or *filename* points to an empty string, a call to *fopen*(*filename*, *type*) returns a **NULL** pointer and sets *errno* to [ENOENT].

32(B) When the directory or file system that would contain the new file cannot be extended, then a call to *fopen*(*filename*, *type*) returns a **NULL** pointer, sets *errno* to [ENOSPC], and the file is not created.

See Reason 1 in Section 5. of POSIX.3 {4}.

33(A) When a component of the *filename* prefix is not a directory, then a call to *fopen*(*filename*, *type*) returns a **NULL** pointer and sets *errno* to [ENOTDIR].

34(C) If the implementation supports a read-only file system:

When the named file resides on a read-only file system, and when *type* is “w”, “a”, “r+”, “w+”, “a+”, “wb”, “ab”, “r+b”, “rb+”, “w+b”, “wb+”, “a+b”, or “ab+”, then a call to *fopen*(*filename*, *type*) returns a **NULL** pointer and sets *errno* to [EROFS].

Testing Requirements:

Test for instances

— When the file does not exist and *type* is “w”, “a”, “w+”, “a+”, “wb”, “ab”, “wb+”, “ab+”, “w+b”, “a+b”, that the file is not created, and

— When the file exists and *type* is “w”, “w+”, “wb”, “wb+”, or “w+b”, that the file is not truncated

8.1.15 *fputc*() (8.1)

8.1.15.1 Synopsis (8.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<stdio.h>` is included, then the function prototype `int fputc(int, FILE *)` is declared. (See GA36 in 2.7.3)

Otherwise:

When the header `<stdio.h>` is included, then the function *fputc*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *fputc*() is defined as a macro when the header `<stdio.h>` is included:

When the macro *fputc*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *fputc*() is defined as a macro in the header `<stdio.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

8.1.15.2 Description (8.2.3.6)

04(A) On a call to *fputc*() the *st_ctime* and *st_mtime* fields of the file are marked for update when data is caused to be written.

Testing Requirements:

Test for the cases when a call to *fputc*() is followed by calls to *fflush*(), *fclose*(), *exit*(), and *abort*().

8.1.15.3 Returns (8.2.3.11)

R01 When a call to *fputc*() completes unsuccessfully, then a value of EOF is returned and sets *errno* to indicate the error. (See Assertions 5-11 in 8.1.15.4)

8.1.15.4 Errors (8.2.3.11)

05(A) When the underlying file descriptor references a pipe or a FIFO that has the O_NONBLOCK flag set and contains insufficient capacity to accept the buffered data, then a call to *fputc*() returns a value of EOF and sets *errno* to [EAGAIN].

Testing Requirements:

Test for both a pipe and a FIFO.

06(A) When the stream pointer argument addresses a file descriptor that is not open for writing, then a call to *fputc*() returns a value of EOF and sets *errno* to [EBADF].

07(A) When *fputc*() is terminated due to receipt of a signal, and when no data was transferred, then the call to *fputc*() returns a value of EOF and sets *errno* to [EINTR].

08(B) When attempting to write to a file that exceeds an implementation-defined maximum file size, then a call to *fputc*() returns a value of EOF and sets *errno* to [EFBIG].

See Reason 3 in Section 5. of POSIX.3 {4}.

09(B) When attempting to write to a device that has no more space for data, then a call to *fputc*() returns a value of EOF and sets *errno* to [ENOSPC].

See Reason 1 in Section 5. of POSIX.3 {4}.

10(A) When attempting to write to a pipe or a FIFO that is not open for reading by any process, then a call to *fputc*() returns a value of EOF, sets *errno* to [EPIPE]. and sends a SIGPIPE signal to the calling process.

Testing Requirements:
Test for both a pipe and a FIFO.

11(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:
When the process is neither ignoring nor blocking the SIGTTOU signal and is a member of an orphaned background process group, when TOSTOP is set, and when the *stream* referenced is the controlling terminal, then a call to *fputc*(*stream*) returns a value of EOF and sets *errno* to [EIO].

8.1.16 *fputs*() (8.1)

8.1.16.1 Synopsis (8.1)

01(A) If the implementation provides C Standard {2} support:
When the header `<stdio.h>` is included, then the function prototype `int fputs(const char *, FILE *)` is declared. (See GA36 in 2.7.3)

Otherwise:
When the header `<stdio.h>` is included, then the function *fputs*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3)

02(C) If *fputs*() is defined as a macro when th header `<stdio.h>` is included:
When the macro *fputs*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3)

03(C) If *fputs*() is defined as a macro in the header `<stdio.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4)

8.1.16.2 Description (8.2.3.6)

04(A) On a call to *fputs*(), the *st_ctime* and *st_mtime* fields of the file are marked for update between its successful completion and the nest successful completion of a call to either *fflush*() or *fclose*() on the same stream or a call to *exit*() or *abort*().

8.1.16.3 Returns (8.2.3.11)

R01 When a call to *fputs*() completes unsuccessfully, then a value of EOF is returned and sets *errno* to indicate the error. (See Assertions 5-11 in 8.1.16.4)

8.1.16.4 Errors (8.2.3.11)

05(A) When the underlying file descriptor references a pipe or a FIFO that has the O_NONBLOCK flag set and contains insufficient capacity to accept the buffered data, then a call to *fputs*() returns a value of EOF and sets *errno* to [EAGAIN].

Testing Requirements:
Test for both a pipe and a FIFO.

06(A) When the stream pointer argument addresses a file descriptor that is not open for writing, then a call to *fputs*() returns a value of EOF and sets *errno* to [EBADF]

07(A) When *fputs*() is terminated due to receipt of a signal, and when no data was transferred, then the call to *fputs*() returns a value of EOF and sets *errno* to [EINTR].

08(B) When attempting to write to a file that exceeds an implementation defined maximum file size, then a call to *fputs* () returns a value of EOF adn sets *errno* to [EFBIG].

See Reason 3 in Section 5. of POSIX.3 {4}.

09(B) When attempting to write to a device that has no more space for data, then a call to *fputs*() returns a value of EOF and sets *errno* to [ENOSPC].

See Reason 1 in Section 5. of POSIX.3 {4}.

10(A) When attempting to write to a pipe or a FIFO that is not open for reading by any process, then a call to *fputs*() returns a value of EOF, sets *errno* to [EPIPE], and sends a SIGPIPE signal to the calling process.

Testing Requirements:

Test for both a pipe and a FIFO.

11(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When the process is neither ignoring nor blocking the SIGTTOU signal and is a member of an orphaned background process group, when TOSTOP is set, and when the stream referenced is the controlling terminal, then a call to *fputs*(*s*, *stream*) returns a value of EOF and sets *errno* to [EIO].

8.1.17 *fread*() (8.1)

8.1.17.1 Synopsis (8.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<stdio.h>` is included, then the function prototype `size_t fread(void *,size_t, size_t, FILE *)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the headr `<stdio.h>` is included, then the function *fread*() is declared with the result type *size_t*. (See GA36 in 2.7.3.)

02(C) If *fread*() is defined as a macro when the header `<stdio.h>` is included:

When the macro *fread*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *size_t*. (See GA37 in 2.7.3)

03(C) If *fread*() is defined as a macro in the header `<stdio.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4)

8.1.17.2 Description (8.2.3.5)

04(A) On a call to *fread*() when data was not supplied by a prior call to *ungetc*(), and when there has been no previous buffered read operation on the stream, then the *st_atime* field of the underlying file is marked for update.

8.1.17.3 Returns (8.2.3.11)

R01 When a call to *fread*() completes unsuccessfully, then the count returned is less than requested and sets *errno* to indicate the error. (See Assertions 5-9 in 8.1.17.4.)

8.1.17.4 Errors (8.2.3.11)

05(A) When the underlying file descriptor references a pipe or a FIFO that has the O_NONBLOCK flag set and contains insufficient data to process a read request, then a call to *fread*() returns a value less than requested and sets *errno* to [EAGAIN].

Testing Requirements:
Test for both a pipe and a FIFO.

06(A) When the stream pointer argument addresses a file descriptor that is not open for reading, then a call to *fread*() returns a value of (*size_t*)0 and sets *errno* to [EBADF].

07(A) When *fread*() is terminated due to receipt of a signal, and when no data was transferred, then the call to *fread*() returns a value of (*size_t*)0 and sets *errno* to [EINTR].

08(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:
When the process is ignoring or blocking the SIGTTIN signal and is a member of a background process group, and when the *stream* argument references the controlling terminal, then a call to *fread*(*ptr*, *size*, *nmemb*, *stream*) returns a value of (*size_t*)0 and sets *errno* to [EIO].

09(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:
When the process is a member of an orphaned background process group, and when the *stream* argument references the controlling terminal, then a call to *fread*(*ptr*, *size*, *nmemb*, *stream*) returns a value of (*size_t*)0 and sets *errno* to [EIO].

8.1.18 *freopen*() (8.1)

8.1.18.1 Synopsis (8.1)

01(A) If the implementation provides C Standard {2} support:
When the header `<stdio.h>` is included, then the function prototype `FILE * freopen(const char *, const char *, FILE *)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<stdio.h>` is included, then the function *freopen*() is declared with the result type *FILE**. (See GA36 in 2.7.3.)

02(C) If *freopen*() is defined as a macro when the header `<stdio.h>` is included:
When the macro *freopen*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *FILE* *. (See GA37 in 2.7.3.)

03(C) If *freopen*() is defined as a macro in the header `<stdio.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

8.1.18.2 Description (8.2.3.3)

04(A) When the first filename component of the *filename* argument is ".", and the pathname does not begin with a slash, then *freopen*() resolves the pathname by locating the second filename component (when specified) in the current working directory. (See GA14 in 2.3.6.)

05(A) When the *filename* argument points to the string "`/`", then *freopen*() resolves the pathname to the root directory of the process. (See GA15 in 2.3.6.)

06(A) When the *filename* argument points to the string "`///`", then *freopen*() resolves the pathname to the root directory of the process. (See GA16 in 2.3.6.)

07(A) When the *filename* argument points to a string beginning with a single slash or beginning with three or more slashes, then *freopen*() resolves the pathname by locating the first filename component of the pathname in the root directory of the process. (See GA17 in 2.3.6.)

08(A) When the first filename component of the *filename* argument is "..", the pathname does not begin with a slash, and the current working directory is not the root directory of the process, then *freopen*() resolves the pathname by locating the second filename component (when specified) in the parent directory of the current working directory. (See GA18 in 2.3.6.)

09(A) When the *filename* argument points to the string "F1/" and `F1` is a directory, then *freopen*() resolves the pathname to `F1`, which is in the current working directory. (See GA19 in 2.3.6.)

10(A) When the argument *filename* points to the string "F1//" and `F1` is a directory, then *freopen*() resolves the pathname to `F1`, which is in the current working directory. (See GA20 in 2.3.6.)

11(A) When the *filename* argument points to the string "F1/F2", then *freopen*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA21 in 2.3.6.)

12(A) When the pathname argument points to the string "F1/./F2", then *freopen*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA22 in 2.3.6.)

13(A) When the *filename* argument points to the string "F1/../F1/F2", then *freopen*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA23 in 2.3.6.)

14(A) When the *filename* argument points to the string "F1//F2", then *freopen*() resolves the pathname to the file `F2` in the directory `F1`, which is in the current working directory. (See GA24 in 2.3.6.)

15(C) If {_POSIX_NO_TRUNC} is not supported in the corresponding directory:

When the *filename* component is a string of more than {NAME_MAX} bytes in a directory for which {_POSIX_NO_TRUNC} is not supported, then *freopen*() resolves the pathname component by truncating it to {NAME_MAX} bytes. (See GA25 in 2.3.6.)

16(A) On a call to *freopen*(*filename*, *mode*, *stream*), the pathname *filename* supports *filenames* containing any of the following characters:

ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789._-

(See GA02 in 2.2.2.32.)

17(A) On a call to *freopen*(*filename*, *mode*, *stream*), the pathname *filename* retains the unique identity of its upper- and lowercase letters. (See GA03 in 2.2.2.32.)

18(A) A call to *freopen*() allocates a file descriptor as a call to *open*() does.

19(A) The *freopen*() function deallocates the file descriptor underlying the stream indicated by its argument (i.e., this file descriptor will be made available for subsequent *open*()s by the process).

20(A) When a call to *freopen*() on the stream is performed, then all outstanding record locks owned by the process on the file associated with the stream are removed.

R01 When a call to *freopen*() is interrupted by a signal that is to be caught, then the call to *freopen*() returns a value of (*int*).1 and sets *errno* to [EINTR]. (See Assertion 32 in 8.1.18.4.)

21(B) When all file descriptors associated with a pipe have been closed by *freopen*(), then data remaining in the pipe is discarded.

See Reason 3 in Section 5. of POSIX.3 {4}.

22(A) When all file descriptors associated with a FIFO special file have been closed by *freopen*(), then data remaining in the FIFO is discarded.

23(B) When all file descriptors associated with an open file description have been closed by *freopen*(), then the open file description is freed.

See Reason 1 in Section 5. of POSIX.3 {4}.

24(B) When all file descriptors associated with a file have been closed by *freopen*(), and when the link count of the file is zero, then the space occupied by the file is freed.

See Reason 1 in Section 5. of POSIX.3 {4}.

25(B) When all file descriptors associated with a file have been closed by *freopen*(), and when the link count of the file is zero, then the file is no longer accessible.

See Reason 3 in Section 5. of POSIX.3 {4}.

26(A) When a file is closed via *freopen*() by the last process that had it open, all time-related fields still marked for update are updated.

27(A) When the referenced stream is writable, and when buffered data has not yet been written, then a call to *freopen*(*filename*, *mode*, *stream*) marks for update the *st_ctime* and *st_mtime* fields of the underlying file.

28(A) When a call to *freopen*() creates a file, then the file permission bits are set to allow both reading and writing for owner, for group and for other users except for those bits set in the file mode creation mask of the process. No execute (search) permission bits are set.

8.1.18.3 Returns (8.2.3.11)

R02 When a call to *freopen*() completes unsuccessfully, then a **NULL** pointer is returned and sets *errno* to indicate the error. (See Assertions 29-33, 35, 38-43 in 8.1.18.4.)

8.1.18.4 Errors (8.2.3.11)

29(A) When search permission is denied on a component of the *filename* prefix, then a call to *freopen*(*filename*, *mode*, *stream*) returns a **NULL** pointer and sets *errno* to [EACCES].

30(A) When the file exists and the requested read and/or write permission is denied, then a call to *freopen*(*filename*, *mode*, *stream*) returns a **NULL** pointer and sets *errno* to [EACCES].

Testing Requirements:

Test for instances when *mode* is “w”, “w+”, “wb”, “wb+”, or “w+b”that the file is not truncated.

31(A) When the file does not exist, and when write permission is denied for the parent directory of the file to be created, then a call to *freopen*(*filename*, *mode*, *stream*) returns a **NULL** pointer, sets *errno* to [EACCES], and the file is not created.

32(A) When *freopen*() is interrupted by a signal, then the call to *freopen*(*filename*, *mode*, *stream*) returns a **NULL** pointer, sets *errno* to [EINTR], and does not mark for update *st_ctime* and *st_mtime* fields of the file and parent directory and the *st_atime* field of the file.

33(A) When the named file is a directory, and when *mode* is “w”, “a”, “r+”, “w+”, “a+”, “wb”, “ab”, “r+b”, “rb+”, “w+b”, “wb+”, “a+b”, or “ab+”, then a call to *freopen*(*filename*, *mode*, *stream*) returns a **NULL**, pointer, sets *errno* to [EISDIR], and does not mark for update *st_ctime*, *st_mtime*, and *st_atime* fields of the file.

34(B) If {OPEN_MAX} ≤ {PCTS_OPEN_MAX}:

When {OPEN_MAX} file descriptors have been opened, then a subsequent call to *freopen*(*filename*, *mode*, *stream*) returns a **NULL** pointer and sets *errno* to [EMFILE].

Testing Requirements:

Test for instances

— When the file does not exist and *mode* is “w”, “a”, “w+”, “a+”, “wb”, “ab”, “wb+”, “ab+”, “w+b”, or “a+b”, that the file is not created, and
— When the file exists and *mode* is “w”, “w+” , “wb” , “wb+”, or “w+b”, that the file is not truncated

Otherwise:

{PCTS_OPEN_MAX} file descriptors can be opened.

See Reason 1 in Section 5. of POSIX.3 {4}.

35({NAME_MAX}≤{PCTS_NAME_MAX}?C: UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the underlying file:

When the length of a pathname component of *filename* is longer than the maximum number of bytes in a filename {NAME_MAX}, then a call to *freopen*(*filename*, *mode*, *stream*) returns a **NULL** pointer and sets *errno* to [ENAMETOOLONG].

36({NAME_MAX}>{PCTS_NAME_MAX}?C :UNTESTED)

If the behavior associated with {_POSIX_NO_TRUNC} is supported for the underlying file:

When the length of a pathname component of *filename* equals {PCTS_NAME_MAX}, then a call to *freopen*(*filename*, *mode*, *stream*) succeeds.

37(A) If {PATH_MAX} ≤ {PCTS_PATH_MAX}:

When the length of *filename* argument exceeds the maximum number of bytes in a pathnAme {PATH_MAX}, then a call to *freopen*(*filename*, *mode*, *stream*) returns a **NULL** pointer and sets *errno* to [ENAMETOOLONG].

Otherwise:

A filename of {PCTS_PATH_MAX} can be opened.

38(B) When too many files are open in the system at the point at which *freopen*() tries to allocate a new file descriptor, then a call to *freopen*(*filename*, *mode*, *stream*) returns a **NULL** pointer and sets *errno* to [ENFILE].

Testing Requirements:

Test for instances

— When the file does not exist and *mode* is “w”, “a”, “w+”, “a+”, “wb ”, “ab”, “wb+”, “ab+”, “w+b”, or “a+b”, that the file is not created, and
— When the file exists and *mode* is “w”, “w+”, “wb”, “wb+”, or “w+b”, that the file is not truncated

See Reason 1 in Section 5. of POSIX.3 {4}.

39(A) When the *mode* is “r”, “r+”, “rb”, “r+b”, or “rb+”, and when the named file does not exist, then a call to *freopen*(*filename*, *mode*, *stream*) returns a **NULL** pointer and sets *errno* to [ENOENT].

40(A) When a component of the path prefix of *filename* does not exist or *filename* points to an empty string, then a call to *freopen*(*filename*, *mode*, *stream*) returns a **NULL** pointer and sets *errno* to [ENOENT].

41(B) When the directory or file system that would contain the new file cannot be extended, then a call to *freopen*(*filename*, *mode*, *stream*) returns a **NULL** pointer, sets *errno* to [ENOSPC], and the file is not created.

See Reason 1 in Section 5. of POSIX.3 {4}.

42(A) When a component of the *filename* prefix is not a directory, then a call to *freopen*(*filename*, *mode*, *stream*) returns a **NULL** pointer and sets *errno* to [ENOTDIR].

43(C) If the implementation supports a read-only file system:

When the named file resides on a read-only file system, and when *mode* is “w”, “a”, “r+”, “w+”, “a+”, “wb”, “ab”, “r+b”, “rb+”“, “w+b”, “wb+”, “a+b”, or “ab+”, then a call to *freopen*(*filename*, *mode*, *stream*) returns 8 **NULL** pointer and sets *errno* to [EROFS].

Testing Requirements:

Test for instances

— When the file does not exist and *mode* is “w”, “a”, “w+”, “a+”, “wb”, “ab”, “wb+”, “ab+”, “w+b”, or “a+b”, that the file is not created, and
— When the file exists and *mode* is “w”, “w+”, “wb”, “wb+”, or “w+b”, that the file is not truncated

8.1.19 *fseek*() (8.1)

8.1.19.1 Synopsis (8.1)

01(A) If the implementation provides C Standard {2} support:
When the header `<stdio.h>` is included, then the function prototype `int fseek(FILE *, long int, int)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<stdio.h>` is included, then the function *fseek*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *fseek*() is defined as a macro when the header `<stdio.h>` is included:
When the macro *fseek*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *fseek*() is defined as a macro in the header `<stdio.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

8.1.19.2 Description (8.2.3.7)

R01 When the file is capable of seeking, and when the file offset has been set beyond the end of existing data in the file by a call to *fseek*() followed by a call to *write*(), then data read from gaps in the file (areas where no data has been written) appear as though bytes with the value zero had been written. (See Assertion 18 in 9.5.2.)

04(A) When the file is capable of seeking, and when buffered data written by *fseek*(), if any, does not extend the file, then a call to *fseek*() does not, by itself, extend the size of the file.

05(A) When the stream is writable and buffered data had not yet been written, then a call to *fseek*() marks for update the *st_ctime* and *st_mtime* fields of the underlying file.

06(A) When a call to *fflush*() was the most recent operation [excluding *ftell*()], then a call to *fseek*() adjusts the file offset in the underlying open file description to reflect the location specified by the *fseek*() call.

8.1.19.3 Returns (8.2.3.11)

R02 When a call to *fseek*() completes unsuccessfully, then a nonzero value is returned and *errno* is set to indicate the error. (See Assertions 7-11 in 8.1.19.4)

8.1.19.4 Errors (8.2.3.11)

07(A) When the *stream* pointer argument addresses a file descriptor that is closed, then the call to *fseek*(*stream*, *offset*, *whence*) sets *errno* to [EBADF] and returns a nonzero value.

08(D) If the implementation provides buffered streams:
When *fseek*() is terminated due to receipt of a signal, and when no data was transferred, then the call to *fseek*() sets *errno* to [EINTR] and returns a nonzero value.

See Reason 3 in Section 5. of POSIX.3 {4}.

09(D) If the implementation provides buffered streams:
When buffered data had not yet been written, and when an attempt to write the data will exceed an implementation-defined maximum file size, then a call to *fseek*() sets *errno* to [EFBIG] and returns a nonzero value.

See Reason 3 in Section 5. of POSIX.3 {4}.

10(D) If the implementation provides buffered streams:
When buffered data had not yet been written, and when an attempt is made to write to a device that has no more space for data, then a call to *fseek*() sets *errno* to [ENOSPC] and returns a nonzero value.

See Reason 1 in Section 5. of POSiX.3 {4}.

11(A) When the underlying open file descripton references a pipe or FIFO, then a call to *fseek*() sets *errno* to [ESPIPE], returns a nonzero value, and the value of the file pointer is not changed.

Testing Requirements:
Test for both a pipe and a FIFO.

8.1.20 *ftell*() (8.1)

8.1.20.1 Synopsis (8.1)

01(A) If the implementation provides C Standard {2} support:
When the header `<stdio.h>` is included, then the function prototype `long int ftell(FILE *)` is declared. (See GA36 in 2.7.3.)

Otherwise:
When the header `<stdio.h>` is included, then the function *ftell*() is declared with the result type *long int*. (See GA36 in 2.7.3.)

02(C) If *ftell*() is defined as a macro when the header `<stdio.h>` is included:
When the macro *ftell*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *long int*. (See GA37 in 2.7.3.)

03(C) If *ftell*() is defined as a macro in the header `<stdio.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

8.1.20.2 Description (8.2.3.10)

04(A) The result of a call to *ftell*() after an *fflush*() is the same as the result before the *fflush*().

8.1.20.3 Returns (8.2.3.11)

R01 When a call to *ftell*() completes unsuccessfully, then a value of (*long*)-1 is returned and sets *errno* to indicate the error. (See Assertions 5 and 6 in 8.1.20.4.)

8.1.20.4 Errors (8.2.3.11)

05(A) When the stream pointer argument addresses a file descriptor that is closed, then the call to *ftell*() returns a value of (*long*)-1 and sets *errno* to [EBADF].

06(A) When the underlying open file description references a pipe or a FIFO, then the call to *ftell*() returns a value of (*long*)-1 and sets *errno* to [ESPIPE].

Testing Requirements:

Test for both a pipe and a FIFO.

8.1.21 *fwrite*() (8.1)

8.1.21.1 Synopsis (8.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<stdio.h>` is included, then the function prototype `size_t fwrite(const void *, size_t, size_t, FInE *)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<stdio.h>` is included, then the function *fwrite*() is declared with the result type *size_t*. (See GA36 in 2.7.3.)

02(C) If *fwrite*() is defined as a macro when the header `<stdio.h>` is included:

When the macro *fwrite*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *size_t*. (See GA37 in 2.7.3.)

03(C) If *fwrite*() is defined as a macro in the header `<stdio.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

8.1.21.2 Description (8.2.3.6)

04(A) On a call to *fwrite*(), the *st_ctime*and *st_mtime*fields of the file are marked for update between its successful completion and the next successful completion of a call to either *fflush*() or *fclose*() on the same stream or a call to *exit*()or *abort*().

05({PIPE_BUF}≤{PCTS_PIPE_BUF}?A:UN TESTED)

When *fwrite*() writes greater than zero bytes but fewer than requested, then the error indicator for the stream is set.

8.1.21.3 Returns (8.2.3.11)

R01 When a call to *fwrite*() completes unsuccessfully, and when no bytes are written, then a value of zero is returned and *errno* is set to indicate the error. (See Assertions 6-12 in 8.1.21.4.)

8.1.21.4 Errors (8.2.3.11)

06(A) When the underlying file descriptor references a pipe era FIFO that has the O_NONBLOCK flag set and contains insufficient capacity to accept any of the buffered data, then a call to *fwrite*() sets *errno* to [EAGAIN]and returns a value of zero.

Testing Requirements:

Test for both a pipe and a FIFO.

07(A) When the stream pointer argument addresses a file descriptor that is not open for writing, then a call to *fwrite*() sets *errno* to [EBADF] and returns a value of zero.

08(A) When *fwrite*() is terminated due to receipt of a signal, and when no data was transferred, then the call to *fwrite*() sets *errno* to [EINTR] and returns a value of zero.

09(B) When attempting to write to a file that exceeds an implementation-defined maximum file size, and when no bytes are written to the file, then a call to *fwrite*() sets *errno* to [EFBIG] and returns a value of zero.

See Reason 3 in Section 5. of POSIX.3 {4}.

10(B) When attempting to write to a device that has no more space for data, and when no bytes are written to the file, then a call to *fwrite*() sets *errno* to [ENOSPC] and returns a value of zero.

See Reason 1 in Section 5. of POSIX.3 {4}.

11(A) When attempting to write to a pipe or a FIFO that is not open for reading by any process, then a call to *fwrite*() sets *errno* to [EPIPE], returns a value of zero, and sends a SIGPIPE signal to the calling process.

Testing Requirements:

Test for both a pipe and a FIFO.

12(PCTS_GTI_DEVICE?C :UNTESTED)

If the behavior associated with {POSIX_JOB_CONTROL} is supported:

When the process is neither ignoring nor blocking the SIGTTOU signal and is a member of an orphaned background process group, when TOSTOP is set, and when the *stream* referenced is the controlling terminal, then a call to *fwrite*(*ptr, size, nmemb, stream*) sets *errno* to [EIO] and returns a value of zero.

8.1.22 *getc*() (8.1)

8.1.22.1 Synopsis (8.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<stdio.h>` is included, then the function prototype `int getc(FILE *)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<stdio.h>` is included, then the function *getc*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *getc*() is defined as a macro when the header `<stdio.h>` is included:

When the macro *getc*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *getc*() is defined as a macro in the header `<stdio.h>`:

It evaluates its argument one or more times, fully protected by parentheses when necessary, and protects its results value with extra parentheses when necessary.

8.1.22.2 Description (8.2.3.5)

04(A) When data was not supplied by a prior call to *ungetc*(), and when there has been no previous buffered read operation on the stream, then a call to *getc*() marks for update the *st_atime* field of the underlying file.

8.1.22.3 Returns (8.2.3.11)

R01 When a call to *getc*() completes unsuccessfully, then a value of EOF is returned and *sets errno* to indicate the error. (See Assertions 5-9 in 8.1.22.4.)

8.1.22.4 Errors (8.2.3.11)

05(A) When the underlying file descriptor references a pipe or a FIFO that has the O_NONBLOCK flag set and contains insufficient data to process a read request, then a call to *getc*() returns a value of EOF and sets *errno* to [EAGAIN].

Testing Requirements:

Test for both a pipe and a FIFO.

06(A) When the stream pointer argument addresses a file descriptor that is not open for reading, then a call to *getc*() returns a value of EOF and sets *errno* to [EBADF].

07(A) When *getc*() is terminated due to receipt of a signal, and when no data was transferred, then the call to *getc*() returns a value of EOF and sets *errno* to [EINTR]

08(C) If the behavior associated with {POSIX_JOB_CONTROL}is supported:

When the process is ignoring or blocking the SIGTTIN signal and is a member of a background process group, and when the *stream*argument references the controlling terminal, then a call to *getc*(*stream*) returns a value of EOF and sets *errno* to [EIO].

09(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When the process is a member of an orphaned background process group, and when the *stream*argument references the controlling terminal, then a call to *getc*(*stream*) returns a value of EOF and sets *errno* to [EIO].

8.1.23 *getchar*() (8.1)

8.1.23.1 Synopsis (8.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<stdio.h>` is included, then the function prototype `int getchar(void)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<stdio.h>` is included, then the function *getchar*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *getchar*() is defined as a macro when the header `<stdio.h>` is included:

When the macro *getchar*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *getchar*() is defined as a macro in the header `<stdio.h>`:

It protects its return value with extra parentheses when necessary. (See GA01 in 1.3.4.)

8.1.23.2 Description (8.2.3.5)

04(A) On a call to *getchar*() when data was not supplied by a prior call to *ungetc*(), and when there has been no previous buffered read operation on the stream, then the *st_atime* field of the underlying file is marked for update.

8.1.23.3 Returns (8.2.3.11)

R01 When a call to *getchar*() completes unsuccessfully, then a value of EOF is returned and sets *errno* to indicate the error. (See Assertions 5-9 in 8.1.23.4.)

8.1.23.4 Errors (8.2.3.11)

05(A) When the underlying file descriptor references a pipe or a FIFO that has the O_NONBLOCK flag set and contains insufficient data to process a read request, then a call to *getchar*() returns a value of EOF and sets *errno* to [EAGAIN[.

Testing Requirements:

Test for both a pipe and a FIFO.

06(A) When *stdin* addresses a file descriptor that is not open for reading, then a call to *getchar*() returns a value of ECF and sets *errno* to [EBADF].

07(A) When *getchar*() is terminated due to receipt of a signal, and when no data was transferred, then the call to *getchar*() returns a value of EOF and sets *errno* to [EINTR].

08(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When the process is ignoring or blocking the SIGTTIN signal, is a member of a background process group, and is attempting to read from its controlling terminal, then a call to *getchar*() returns a value of EOF and sets *errno* to [EIO].

09(PCTS_GTI_DEVICE?C :UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When the process is a member of an orphaned background process group and is attempting to read from its controlling terminal, then a call to *getchar*() returns a value of EOF and sets *errno* to [EIO].

8.1.24 *gets*() (8.1)

8.1.24.1 Synopsis (8.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<stdio.h>` is included, then the function prototype `char * gets(char *)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<stdio.h>` is included, then the function *gets*() is declared with the result type *char* *. (See GA36 in 2.7.3.)

02(C) If *gets*() is defined as a macro when the header `<stdio.h>` is included:

When the macro *gets*() is invoked with the correct argument types(or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *char* *. (See GA37 in 2.7.3.)

03(C) If *gets*() is defined as a macro in the header `<stdio.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

8.1.24.2 Description (8.2.3.5)

04(A) When data was not supplied by a prior call to *ungetc*(), and when there has been no previous buffered read operation on the stream, then a call to *gets*() marks for update the *st_atime* field of the underlying file.

8.1.24.3 Returns (8.2.3.11)

R01 When a call to *gets*() completes unsuccessfully, then a **NULL** pointer is returned and sets *errno* to indicate the error. (See Assertions 5-9 in 8.1.24.4.)

8.1.24.4 Errors (8.2.3.11)

05(A) When the underlying file descriptor references a pipe or a FIFO that has he O_NONBLOCK flag set and contains insufficient data to process a read request, then a call to *gets*() returns a **NULL** pointer and sets *errno* to [EAGAIN].

Testing Requirements:

Test for both a pipe and a FIFO.

06(A) When *stdin* addresses a file descriptor that is not open for reading, then a call to *gets*() returns a **NULL** pointer and sets *errno* to [EBADF].

07(A) When *gets*() is terminated due to receipt of a signal, and when no data was transferred, then the call to *gets*() returns a **NULL** pointer and sets *errno* to [EINTR].

08(C) If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When the process is ignoring or blocking the SIGTTIN signal and is a member of a background process group, and when stdin references the controlling terminal, then a call to *gets*(*buf*) returns a **NULL** pointer and sets *errno* to [EIO].

09(PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_ JOB_CONTROL} is supported:

When the process is a member of an orphaned background process group, and when stalin references the controlling terminal, then a call to *gets*(*buf*) returns a **NULL** pointer and sets *errno* to [EIO].

8.1.25 *perror*() (8.1)

8.1.25.1 Synopsis (8.1)

01(C) If the implementation provides C Standard {2} support:

When the header `<stdio.h>` is included, then the function prototype `void perror(const char *)` is declared. (See GA36 in 2.7.3.)

D01(C) If the implementation provides Common-Usage C support:

The result type for function *perror*() is contained in 8.1 of the PCD.1. (See DGA01 in 1.3.3.3.)

02(D) If the implementation provides C Standard {2} support and *perror*() is defined as a macro when the header `<stdio.h>` is included:

When the macro *perror*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *void*. (See GA37 in 2.7.3.)

See Reason 2 in Section 5, of POSIX.3 {4}.

03(C) If *perror*() is defined as a macro in the header `<stdio.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary. (See GA01 in 1.3.3.)

8.1.25.2 Description (8.2.3.8)

04(A) A call to *perror*() marks for update the *st_ctime* and *st_mtime* fields of the underlying file associated with the standard error stream at some time between its successful completion and the completion of *fflush*() or *fclose*() on stderr, or *exit*() or *abort*().

8.1.25.3 Returns (8.2.3.11)

There are no assertions specific to this subclause.

8.1.25.4 Errors (8.2.3.11)

There are no assertions specific to this subclause.

8.1.26 *printf*() (8.1)

8.1.26.1 Synopsis (8.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<stdio.h>` is included, then the function prototype `int printf(const char *, ...)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<stdio.h>` is included, then the function *printf*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *printf*() is defined as a macro when the header `<stdio.h>` is included:

When the macro *printf*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *printf*() is defined as a macro in the header `<stdio.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

8.1.26.2 Description (8.2.3.6)

04(A) On a call to *printf*(), the *st_ctime* and *st_mtime* fields of the file are marked for update between its successful completion and the next successful completion of a call to either *fflush*() or *fclose*() on stdout or a call to *exit*() or *abort*().

8.1.26.3 Returns (8.2.3.11)

R01 When a call to *printf*() completes unsuccessfully, then a negative value is returned and sets *errno* to indicate the error. (See Assertions 5-11 in 8.1.26.4.)

8.1.26.4 Errors (8.2.3.11)

05(A) When the underlying file descriptor references a pipe or a FIFO that has the O_NONBLOCK flag set and contains insufficient capacity to accept the buffered data, then a call to *printf*() returns a negative value and sets *errno* to [EAGAIN].

Testing Requirements:

Test for both a pipe and a FIFO.

06(A) When *stdout* addresses a file descriptor that is not open for writing,then a call to *printf*() returns a negative value and sets *errno* to [EBADF].

07(A) When *printf*() is terminated due to receipt of a signal, and when no data was transferred, then the call to *printf*() returns a negative value and sets *errno* to [EINTR].

08(B) When attempting to write to a file that exceeds an implementation-defined maximum file size, then a call to *printf*()returns a negative value and sets *errno* to [EFBIG].

See Reason 3 in Section 5. of POSIX.3 {4}.

09(B) When attempting to write to a device that has no more space for data, then a call to *printf*() returns a negative value and sets *errno* to [ENOSPC].

See Reason 1 in Section 5. of POSIX.3 {4}.

10(A) When attempting to write to a pipe or a FIFO that is not open for reading by any process, then a call to *printf*() returns a negative value, sets *errno* to [EPIPE], and sends a SIGPIPE signal to the calling process.

Testing Requirements:

Test for both a pipe and a FIFO.

11 (PCTS_GTI_DEVICE?C:UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When the process is neither ignoring nor blocking the SIGTTOU signal and is a member of an orphaned background process group, when TOSTOP is set, and when stdout is the controlling terminal, then a call to *printf*() returns a negative value and sets *errno* to [EIO].

8.1.27 *fprintf*() (8.1)

8.1.27.1 Synopsis (8.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<stdio .h>` is included, then the function prototype `int fprintf(FILE *, const char *, …)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<stdio.h>` is included, then the function *fprintf*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *fprintf*() is defined as a macro when the header `<stdio.h>` is included:

When the macro *fprintf*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *fprintf*() is defined as a macro in the header `<stdio.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

8.1.27.2 Description (8.2.3.6)

04(A) On a call to *fprintf*(), the *st_ctime* and *st_mtime* fields of the file are marked for update between its successful completion and the next successful completion of a call to either *fflush*() or *fclose*() on the same stream or a call to *exit*() or *abort*().

8.1.27.3 Returns (8.2.3.11)

R01 When a call to *fprintf*() completes unsuccessfully, then a negative value is returned and sets *errno* to indicate the error. (See Assertions 5-11 in 8.1.27.4.)

8.1.27.4 Errors (8.2.3.11)

05(A) When the underlying file descriptor references a pipe or a FIFO that has the O_NONBLOCK flag set and contains insufficient capacity to accept the buffered data, then a call to *fprintf*() returns a negative value and sets *errno* to [EAGAIN].

Testing Requirements:

Test for both a pipe and a FIFO.

06(A) When the stream pointer argument addresses a file descriptor that is not open for writing, then a call to *fprintf*() returns a negative value and sets *errno* to [EBADF].

07(A) When *fprintf*() is terminated due to receipt of a signal, and when no data was transferred, then the call to *fprintf*() returns a negative value and sets *errno* to [EINTR].

08(B) When attempting to write to a file that exceeds an implementation-defined maximum file size, then a call to *fprintf*() returns a negative value and sets *errno* to [EFBIG].

See Reason 3 in Section 5. of POSIX.3 {4}.

09(B) When attempting to write to a device that has no more space for data, then a call to *fprintf*() returns a negative value and sets *errno* to [ENOSPC].

See Reason 1 in Section 5. of POSIX.3 {4}.

10(A) When attempting to write to a pipe or a FIFO that is not open for reading by any process, then a call to *fprintf*() returns a negative value, sets *errno* to [EPIPE], and sends a SIGPIPE signal to the calling process.

Testing Requirements:

Test for both a pipe and a FIFO.

11(PCTS_GTI_DEVICE?C :UNTESTED)

If the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When the process is neither ignoring nor blocking the SIGTTOU signal and is a member of an orphaned background process group, when TOSTOP is set, and when the *stream* referenced is the controlling terminal, then a call to *fprintf*(*stream, format*) returns a negative value and sets *errno* to [EIO].

8.1.28 *sprintf*() (8.1)

8.1.28.1 Synopsis (8.1)

01(A) If the implementation provides C Standard {2} support:

When the header `<stdio.h>` is included, then the function prototype `int sprintf(char *, const char *, …)` is declared. (See GA36 in 2.7.3.)

Otherwise:

When the header `<stdio.h>` is included, then the function *sprintf*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

02(C) If *sprintf*() is defined as a macro when the header `<stdio.h>` is included:

When the macro *sprintf*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

03(C) If *sprintf*() is defined as a macro in the header `<stdio.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

8.1.28.2 Description (8.1)

04(B) If the implementation provides C Standard {2} support:

Element *sprintf*()complies with the requirements of the C Standard {2}. (See GA63 in 8..)

Otherwise:

Element *sprintf*() complies with the requirements of the C Standard {2} unless amended by the PCD.1. (See GA63 in 8..)

See Reason 2 in Section 5. of POSIX.3 {4}.

8.1.28.3 Returns (8.1)

There are no assertions specific to this subclause.

8.1.28.4 Errors (8.1)

There are no assertions specific to this subclause.

8.1.29 *vprintf*() (8.2.3.6)

8.1.29.1 Synopsis (8.2.3.6)

01(C) If the implementation provides C Standard {2} support:
When the header `<stdio.h>` is included, then the function prototype
`int vprintf(const char *, va_list)` is declared. (See GA36 in 2.7.3.)

02(C) If the implementation provides Common-Usage C and *vprintf*() from the C Standard {2}:
When the header `<stdio.h>` is included, then the function *vprintf*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

03(C) If the implementation provides *vprintf*() from the C Standard {2} and *vprintf*() is defined as a macro when the header `<stdio.h>` is included:
When the macro *vprintf*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

04(C) If the implementation provides *vprintf*() from the C Standard {2} and *vprintf*() is defined as a macro in the header `<stdio.h>`:
It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

8.1.29.2 Description (8.2.3.6)

05(C) If the implementation provides *vprintf*() from the C Standard {2}.
On a call to *vprintf*(), the *st_ctime* and *st_mtime* fields of the file are marked for update between its successful completion and the next successful completion of a call to either *fflush*() or *fclose*() on stdout or a call to *exit*() or *abort*().

8.1.29.3 Returns (8.2.3.11)

R01 When a call to *vprintf*() completes unsuccessfully, then a negative value is returned and sets *errno* to indicate the error. (See Assertions 6-12 in 8.1.29.4.)

8.1.29.4 Errors (8.2.3.11)

06(C) If the implementation provides *vprintf*() from the C Standard {2}:
When the underlying file descriptor references a pipe or a FIFO that has the O_NONBLOCK flag set and contains insufficient capacity to accept the buffered data, then a call to *vprintf*() returns a negative value and sets *errno* to [EAGAIN].
Testing Requirements:
Test for both a pipe and a FIFO.

07(C) If the implementation provides *vprintf*() from the C Standard {2}:
When stdout addresses a file descriptor that is not open for writing, then a call to *vprintf*() returns a negative value and sets *errno* to [EBADF].

08(C) If the implementation provides *vprintf*() from the C Standard {2}:
When *vprintf*() is terminated due to receipt of a signal, and when no data was transferred, then the call to *vprintf*() returns a negative value and sets *errno* to [EINTR].

09(D) If the implementation provides *vprintf*() from the C Standard {2}:

When attempting to write to a file that exceeds an implementation-defined maximum file size, then a call to *vprintf*() returns a negative value and sets *errno* to [EFBIG].

See Reason 3 in Section 5. of POSIX.3 {4}.

10(D) If the implementation provides *vprintf*() from the C Standard {2}:

When attempting to write to a device that has no more space for data, then a call to *vprintf*() returns a negative value and sets *errno* to [ENOSPC].

See Reason 1 in Section 5. of POSIX.3 {4}.

11(C) If the implementation provides *vprintf*() from the C Standard {2}:

When attempting to write to a pipe or a FIFO that is not open for reading by any process, then a call to *vprintf*() returns a negative value, sets *errno* to [EPIPE], and sends a SIGPIPE signal to the calling process.

Testing Requirements:

Test for both a pipe and a FIFO.

12(PCTS_GTI_DEVICE?C:UNTESTED)

If the implementation provides *vprintf*() from the C Standard {2} and the behavior associated with {_POSIX_JOB_CONTROL} is supported:

When the process is neither ignoring nor blocking the SIGTTOU signal and is a member of an orphaned background process group, when TOSTOP is set, and when the *stream* referenced is the controlling terminal, then a call to *vprintf*(*stream*, *format*) returns a negative value and sets *errno* to [EIO].

8.1.30 *vfprintf*() (8.2.3.6)

8.1.30.1 Synopsis (8.2.3.6)

01(C) If the implementation provides C Standard {2} support:

When the header `<stdio.h>` is included, then the function prototype `int vfprintf(FILE *, const char *, va_list)` is declared. (See GA36 in 2.7.3.)

02(C) If the implementation provides Common-Usage C and *vfprintf*() from the C Standard {2}:

When the header `<stdio.h>` is included, then the function *vfprintf*() is either declared with result type *int* or not declared in the header. (See GA36 in 2.7.3.)

03(C) If the implementation provides *vfprintf*() from the C Standard {2} and *vfprintf*() is defined as a macro when the header `<stdio.h>` is included:

When the macro *vfprintf*() is invoked with the correct argument types (or compatible argument types in the case that C Standard {2} support is provided), then it expands to an expression with the result type *int*. (See GA37 in 2.7.3.)

04(C) If the implementation provides *vfprintf*() from the C Standard {2} and *vfprintf*() is defined as a macro in the header `<stdio.h>`:

It evaluates its arguments only once, fully protected by parentheses when necessary, and protects its result value with extra parentheses when necessary. (See GA01 in 1.3.4.)

8.1.30.2 Description (8.2.3.6)

05(C) If the implementation provides *vfprintf*() from the C Standard {2}:

On a call to *vfprintf*(), the *st_ctime* and *st_mtime* fields of the file are marked for update between its successful completion and the next successful completion of a call to either *fflush*() or *fclose*() on the same stream or a call to *exit*() or *abort*().

